Banco de tiempo con oferta y demanda de servicios en una red P2P

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2016-2017

Yamal César Al-Mahamid Vélez

Director

Simon Pickin

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Madrid, 15 de junio de 2017

Agradecimientos

A mis padres, por haberme apoyado tanto a lo largo de estos años y por haberme dado la oportunidad de haber llegado hasta aquí. A mi hermano mayor, por haberme ayudado tanto a lo largo de la carrera y por servirme de ejemplo, y a mi hermano pequeño por haber estado siempre a mi lado.

A mi director en este proyecto, el Dr. Simon Pickin, por todo lo que me ha enseñado durante este último año y por haberme ayudado a llevar el desarrollo por buen camino.

A todos mis compañeros de la universidad de los que me llevo muy buenos recuerdos y a todos los que han estado conmigo desde el principio.

A todos, muchas gracias.

Resumen

La tecnología P2P desde su inicio ha vivido etapas de proliferación y otras en las que su avance se ha visto frenado. Las primeras redes P2P surgieron a finales del siglo XX y principalmente tenían como propósito el intercambio de ficheros entre diferentes puntos de la red. Aprovechando esta facilidad en los intercambios, los usuarios de estas aplicaciones las usaban para compartir de manera gratuita contenido sujeto a copyright, por ejemplo ficheros mp3, por lo que las entidades encargadas de defender la propiedad intelectual siempre han luchado por el cierre de estas redes.

Por este motivo siempre se ha relacionado el concepto de piratería con la tecnología P2P, lo que ha provocado que en los últimos años no se hayan conseguido grandes avances en este campo para poder sacar provecho de los beneficios que una solución descentralizada ofrece.

Lo que el presente documento pretende es promover la investigación de soluciones descentralizadas para aplicaciones de utilidad social, no destinadas al intercambio de contenido, mediante el análisis, diseño e implementación prototípica de un banco de tiempo sobre una red P2P. Se quieren buscar alternativas viables para esta aplicación escogida que se puedan trasladar a otras aplicaciones de otros ámbitos pero también sobre tecnología P2P.

El presente proyecto es una continuación de otro TFG [1], el cual consistió en la construcción de un banco de tiempo P2P pero dejó unas partes incompletas e incluso algo incoherentes, y otras partes completamente sin desarrollar. Las correcciones aquí realizadas han mejorado el análisis y diseño de la parte de las transacciones de la aplicación respecto al proyecto previo, y como continuación de su desarrollo se ha realizado el análisis, diseño y una primera implementación del subsistema de oferta y demanda de servicios del banco de tiempo, permitiendo a potenciales consumidores de un servicio encontrar a proveedores del mismo. Para esta nueva parte de la aplicación, la cual no estaba previamente definida, se ha realizado un estudio amplio en el que se ha visto la necesidad de emplear una ontología dinámica, algo que es novedoso, para la cual se ha realizado casi todo el análisis y diseño de su uso en el banco de tiempo.

Abstract

Since its beginnings, P2P technology has lived through boom periods and periods in which progress has been slow. The first P2P networks arose at the end of the 20th century mainly for the purpose of file sharing between different peers. The users of these application took advantage of this ease of file sharing to share copyrighted content, in particular MP3 files, which is why the development and deployment of these networks have been vigorously opposed by companies and organizations trying to protect intellectual property.

This fact has meant that, to the general public, P2P technology is practically synonymous with digital piracy and has lead to lack of progress in taking advantage of the benefits that these decentralized solutions can offer.

The aim of this document is to promote the investigation of decentralized solutions in the creation of socially-useful applications, as opposed to simple file-sharing applications, via the analysis, design and prototype implementation of a time bank application built on top of a P2P network. In undertaking this development, we have sought to define concepts and develop artefacts that could be reused across a range of P2P applications.

The present project is a continuation of a previous TFG, which consisted of the development of a P2P time bank but which left some parts of the application incomplete and even inconsistent, and others completely undefined. The corrections made here have improved the analysis and design of the transactions subsystem of the application, with respect to the previous project. Next, as a continuation of the development, the analysis, design and first implementation of the time bank application subsystem dealing with publication and search of services, enabling potential consumers of a service to find providers of that service, was carried out. For this new part of the application, completely undefined in the previous project, a wide study was undertaken from which the need to employ a dynamic ontology, a relatively novel concept, was observed. The analysis and design of the use of dynamic ontologies in the time bank application was a key part of the analysis and design of this latter subsystem.

Palabras clave

- P2P
- Banco de tiempo
- FreePastry
- DHT
- Ontología dinámica
- Oferta y demanda de servicios
- DAG (Directed Acyclic Graph)
- Combinación de grafos

Keywords

- P2P
- Time bank
- FreePastry
- DHT
- Dynamic Ontology
- Publication and search of services
- DAG (Directed Acyclic Graph)
- Graph merge

Índice

Índice de figuras

1 Introducción

En este apartado se hará una introducción al trabajo realizado para comprender en futuros capítulos los principios básicos en los que se basa. En primer lugar se describirán los antecedentes que han llevado a realizar este proyecto, es decir, el contexto previo y el surgimiento de las ideas. En segundo lugar se explicarán los objetivos que se pretenden cubrir al finalizar el trabajo. Y para terminar, el último punto de esta introducción será detallar el plan de trabajo planteado al inicio del proyecto.

1.1 Antecedentes

Aunque se hablará más adelante de los objetivos del trabajo, la razón de este proyecto es continuar con el desarrollo de una aplicación de utilidad social bajo la tecnología P2P, y esa aplicación de utilidad social es un banco de tiempo. El trabajo inicial [1] en el que se basa el actual fue también un trabajo de fin de grado realizado en el curso 2014/2015 por los alumnos Antonio Núñez Guerrero, Daniel Alejandro Nowendsztern y Marcos Pérez García dirigido por el mismo tutor de este trabajo, Simon Pickin. Su proyecto estuvo centrado casi exclusivamente en la especificación de un banco de tiempo P2P, y el estado final de su trabajo consistió en una especificación de requisitos y funcionalidad de la aplicación: división en módulos del sistema, estado del arte bien documentado, y definición clara del trabajo futuro ya que hubo muchos aspectos que no fueron cubiertos. Asimismo también se llegó a implementar un esqueleto de la aplicación con una estructura bien definida. Los puntos definidos dentro del banco de tiempo se basan principalmente en la información que iba a ser tratada: los ficheros del sistema, información de cada usuario, gestión de las transacciones… Sin embargo, quedaron módulos importantes de la aplicación sin especificar, y eran aquellos en los que la idea de usar tecnología P2P jugaba un papel importante. Es por eso que en uno de los puntos de este trabajo se resumirá el banco de tiempo previo definido por los alumnos que nos precedieron, para entender bien el resto de conceptos referidos a lo largo del proyecto actual.

Antes de seguir es necesario aclarar que en el presente trabajo ha habido en algún punto solapes inevitables respecto al proyecto anterior, sin los que el trabajo carecería de suficiente información. Entre estos puntos se encuentra el estudio del estado del arte de las tecnologías empleadas, la definición de banco de tiempo y otros capítulos en los que se habla del diseño de la aplicación. No obstante, se han intentado minimizar estos solapes ya que todo el proceso de estudio e investigación se ha realizado de manera independiente.

1.2 Objetivos

Aunque se pretende seguir en este trabajo con el diseño e implementación de un banco de tiempo P2P, también se pretende continuar con el estudio que motivó el querer desarrollar una aplicación de este tipo. Esta motivación consistía en investigar diversas

formas de construir soluciones P2P que se pudieran generalizar y usarse en aplicaciones de utilidad social de cualquier ámbito. La búsqueda de estas soluciones se basa en intentar solucionar problemas detectados en el banco de tiempo y especificarlas como algo genérico. Entre estas soluciones se encuentra la definición de un sistema de oferta y demanda descentralizado y gestionado por un conjunto de usuarios pertenecientes a un sistema P2P, como puede ser por ejemplo una plataforma de venta de productos de segunda mano, aparte del banco de tiempo.

A continuación se enumeran todos los objetivos del trabajo:

1. Estudio de las tecnologías P2P.

2. Revisión y corrección del diseño previo del banco de tiempo.

3. Implementación de algunas funcionalidades principales de la aplicación.

4. Investigar nuevas soluciones a problemas y limitaciones en aplicaciones P2P de utilidad social encontradas durante la especificación del banco de tiempo.

1.3 Plan de trabajo

El plan de trabajo seguido se basó en los objetivos que se pretendían cubrir al inicio del trabajo.

A lo largo de las primeras semanas del proyecto tuvo lugar el acercamiento y comprensión de la especificación del banco de tiempo inicial y del trabajo previamente realizado. Durante los meses siguientes se tuvieron que realizar las correcciones y adiciones necesarias a lo que se había hecho de la aplicación: casos de uso, modelo de datos, modelo de dominio y diagrama de secuencia.

Al mismo tiempo que ocurría esto, se realizó un estudio de los sistemas y redes P2P para poder entender sobre qué tipo de tecnologías se estaba trabajando.

El siguiente paso fue buscar una herramienta o framework para construir una primera versión demostrable de una parte de la aplicación, en concreto del protocolo de pago.

Finalmente, en los últimos meses, el trabajo se centró en investigar qué posibles aspectos del banco de tiempo que quedaron sin definir en el proyecto anterior mostraban más interés para ser estudiados. Tras elegir el sistema de oferta y demanda de servicios en el banco de tiempo como aspecto importante a definir, el proyecto le dedicó el resto del tiempo a esta tarea. Y dicha tarea incluyó: estudio de los antecedentes y alternativas vistas en otras aplicaciones, propuesta de posibles soluciones a este problema y análisis de la solución que finalmente se escogió.

2 Introduction

This chapter is an introduction to the work done to understand in the future chapters the basic principles on which it is based. First the background that initiated this project will be described, the previous context and the emergence of the ideas. Secondly the goals we want to reach at the end of the project will be explained. And to finish, the las point of this introduction will be the details of the work plan proposed at the beginning of the project.

2.1 Background

Although we will talk later about the goals of the work, the reason for this project is to continue the development of a social utility application under P2P technology, and this application is a time bank. The initial work [1], on which is based the current one, was a final degree project too performed in the course 2014/2015 by the students Antonio Núñez Guerrero, Daniel Alejandro Nowendsztern and Marcos Pérez, and it was leaded by the same tutor as this year, Simon Pickin. Their project was mainly focused on a P2P time bank specification, and the final state of their work consisted of a requirements and functionality specification: Division of system into modules, state of the art well documented, and a clear definition of the future work as there were a lot of points that were not defined. A frame for the application was implemented too with a well-defined structure. The specified points of the time bank are based in the processed information: system files, each user information, transaction management... Nevertheless, some modules were left unspecified, and they were those modules where using P2P technology played an important role. That is why the time bank of the previous project will be described to understand the rest of the concepts in this document.

Before proceeding, it is necessary to clarify that there have been some inevitable overlaps with the previous work, without which the project would lack of some important content. Among those points we can find the state of art and other chapters about the design of the application. However, we have tried to minimize these overlaps and the whole process of study and investigation has been done independently.

2.2 Goals

Despite we intend to continue with the design and implementation of a P2P time bank, actually we also intend to continue with the study that motivated the desire of developing an application like this. This motivation consisted of investigate some ways of build P2P solutions which could be generalized and used in social applications of different scopes. The search of these solutions is based on trying to solve problems detected in the decentralized time bank and specify them as something generic.

Among these solutions, we distinguish the definition of a decentralized system of publication and search of services managed by a group of users distributed over the P2P network. Apart of a time bank, an example of application which can use this system is a sales platform for second-hand products.

All goals of the project are listed below:

- P2P technologies study.
- Revision and correction of the previous design of the time bank.
- Implementation of some main functionalities of the application.
- Investigation of new possible solutions to problems and limitations of P2P applications detected during the time bank specification.

2.3 Work plan

The work plan followed was based on the goals that we intended to reach at the beginning of the project.

Through the first weeks of the project the oncoming and comprehension of the previous work and the initial time bank specification took place. During the next months we had to do some necessary corrections and additions to the things already done: uses cases, data model, domain model and sequence diagrams.

At the same time, a study of P2P network and systems was carried out to understand what kind of technologies we were working over.

The next step was to find a tool or framework to build a first provable version of a part of the application, in particular the payment protocol.

In the last months, the work was focused on research what aspects of the time bank that were not defined in the past project, showed more interest to be described. After choosing the system of publication and search of services as the more important aspect at that moment to be defined, the project dedicated the rest of the time to this task. That task included: background study, alternatives to this problem seen in others applications, proposal of different solutions and analysis of the solution finally chosen.

3 Estado del arte

En este capítulo se profundizará en todos los aspectos implicados en el proyecto principalmente para que sirva de introducción a los capítulos posteriores, pero al mismo tiempo, la recolección de toda esta información también ha servido como fase de aprendizaje.

A continuación se hablará por un lado de los bancos de tiempo, y por otro lado se tratará la parte relacionada con la tecnología P2P, es decir, la base de este trabajo.

3.1 Bancos de tiempo

3.1.1 ¿Qué es y cómo funciona?

Un banco de tiempo es una plataforma en la que se intercambian servicios a cambio de tiempo. La unidad de intercambio utilizada para ello no es el dinero sino una unidad de medida de tiempo, generalmente el trabajo por horas. La idea principal consiste en realizar un servicio a cambio del tiempo que cueste llevarlo a cabo y a su vez usar este crédito adquirido para demandar otros servicios. Los servicios que se pueden intercambiar abarcan un alto espectro de opciones, desde atención y cuidado de personas hasta formación académica (generalmente de forma particular), pasando por tareas domésticas, conversación en otros idiomas, asesoramiento, etc.

El funcionamiento de los bancos de tiempo es en realidad algo sencillo. Aunque cada banco suele tener sus propias características particulares dependiendo de la entidad que lo gestione, todos siguen la misma idea original [2]:

- Los usuarios se dan de alta en el banco creando una cuenta de horas o de la unidad de tiempo que se haya elegido.
- Cuando un usuario presta un servicio gana horas que se acumularán en su cuenta.
- Con las horas que un usuario ha ido acumulando puede canjearlas y demandar servicios que otros usuarios ofrecen.

3.1.2 Uso de los bancos de tiempo

En general, el concepto de banco de tiempo no está muy extendido entre la mayoría de las personas. No obstante, aun siendo una idea poco conocida, existe una gran red de bancos de tiempo que se extiende cada vez más. Por ejemplo, solo en la Comunidad de Madrid, ya hay más de cuarenta bancos de tiempo [3].

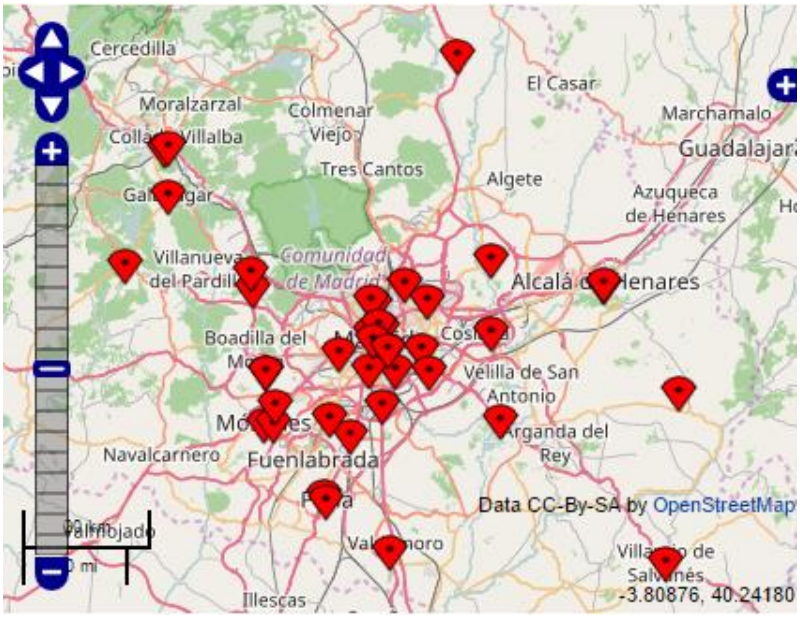

Figura 3.1: Bancos de tiempo en la Comunidad de Madrid

Este tipo de sistemas siempre se han realizado en pequeñas comunidades como barrios o núcleos urbanos dentro de las ciudades, pero de una manera no informatizada. En los últimos años con el continuo crecimiento de Internet y el cada vez más fácil acceso a éste, han aparecido bancos de tiempo gestionados desde la red para facilitar la comodidad de los usuarios y el acceso a la información. Algunos ejemplos de entidades que ofrecen estos servicios a través de una web son el Banco de Tiempo de Manoteras [4] o el Banco de Tiempo A2Manos [5].

Aunque una web o entidad informatizada y centralizada sea actualmente el mejor modo para gestionar eficientemente este tipo de sistemas, requiere un esfuerzo y coste de mantenimiento por parte de las personas que lo organizan, es decir, se necesita mantener toda la infraestructura y todo lo que ello conlleva: diseño web o de la aplicación, gestión de base de datos, seguridad si procede, servidor donde alojar el sitio y los datos, etc. Como un banco de tiempo no genera beneficios económicos ni lo pretende, estas labores de soporte tienen que ser llevadas a cabo por voluntarios. Son necesarias por un lado, personas que compartan sus conocimientos informáticos, y por otro lado, personas que hagan donaciones o aportaciones económicas por ejemplo para pagar el alojamiento del sitio en un servidor. Esta es la razón por la que existen muchos inconvenientes a la hora de diseñar una plataforma virtual para un banco de tiempo. Aunque este problema se puede trasladar a cualquier tipo de actividad similar. Es por eso, que una buena alternativa es usar una red distribuida para repartir esta administración.

3.2 Tecnología P2P

3.2.1 Introducción

3.2.1.1 Qué son las redes P2P

La tecnología P2P (Peer-to-Peer [6]) es un modo de diseñar redes de computadores de tal manera que estos se comportan como iguales entre sí, es decir, que no exista entre ellos la relación cliente-servidor. Los nodos que conforman la red son capaces de intercambiar servicios y recursos sin necesidad de un servidor que centralice las operaciones.

A la hora de diseñar arquitecturas distribuidas, e incluso procesamiento distribuido, elegir una red P2P es una opción que cuenta con muchas ventajas. Una de las cualidades principales de los sistemas P2P es que proporcionan alta escalabilidad. Esto quiere decir que la red es capaz de responder correctamente ante el crecimiento del número de nodos que hay en ella. En general, cuantos más nodos haya conectados a la red, mejor será el funcionamiento de ésta.

Este tipo de sistemas tienen como principal objetivo no depender de un servidor central, por lo que aparte de surgir ventajas, también aparecen inconvenientes. A continuación se detallan las ventajas e inconvenientes de las redes P2P.

3.2.1.2 Ventajas

Escalabilidad

Cuanto mayor sea la cantidad total de recursos en la red, más y mejor acceso tendrán a estos los nodos que la forman. Por consiguiente, como bien se ha mencionado antes, cuanto más grande sea la red, mejor funcionará, puesto que con cada nueva conexión, los recursos de los nuevos nodos se comparten y la cantidad total de recursos aumenta.

Distribución de costes

Al no existir una entidad o servidor central, los costes tales como ancho de banda, almacenamiento o recursos… deben ser compartidos entre los nodos de la red. Así mismo, la administración del sistema se gestiona entre todos.

Robustez

Los sistemas P2P proporcionan una alta robustez puesto que responde bien a los fallos de accesos a información gracias a la réplica de ésta entre los nodos de la red. Como consecuencia, el acceso a la información mejora.

3.2.1.3 Inconvenientes

Las desventajas de este tipo de sistemas tienen lugar en el ámbito de la administración y en todas las tareas que de ella dependen.

La gestión y el mantenimiento de la red recaen en los nodos que la forman. Por eso, es difícil determinar quién tiene el control sobre los recursos de la red y cómo se llevará a cabo. A este problema se le añade el de la **seguridad**. Como no existe una unidad central de control, las medidas de seguridad ya no son globales, sino que cada nodo debe encargarse de la protección de los datos por sí mismo.
Por otro lado, la responsabilidad de realizar las réplicas de los datos o copias de seguridad recae, una vez más, en los nodos de la red, provocando un fenómeno conocido como *churn* en el que los datos a los que se intenta acceder no están disponibles ya que los nodos responsables de ellos no están conectados a la red en ese momento.

Sin embargo, existen diversas soluciones para tratar estos obstáculos en el diseño de una red P2P que se detallarán más adelante.

3.2.2 Arquitecturas P2P

Todos los sistemas P2P comparten la misma idea de trabajar sin una entidad central que controle toda la red. Sin embargo, hay diferentes topologías de red empleadas para implementar un sistema de este tipo. Estas arquitecturas se diferencian entre ellas por el grado de descentralización que exista. Por eso, se clasifican en tres tipos: Centralizadas, descentralizadas e híbridas.

3.2.2.1 Redes centralizadas

Los sistemas P2P centralizados [7] se caracterizan por combinar el concepto de descentralización con el de cliente-servidor.

En primer lugar, existe la noción de centralización porque los *peers* o nodos de la red se comunican con un nodo central que actúa como servidor, al cual realizan peticiones para determinar la dirección de los nodos que poseen el recurso solicitado. Y en segundo lugar, esta arquitectura es también una solución descentralizada ya que una vez realizada la conexión entre dos nodos (gracias al servidor), ya no es necesaria la presencia de este para comunicarse entre ellos.

El hecho de tener un servidor que gestione la resolución de direcciones genera tanto ventajas como inconvenientes. Es decir, la dificultad de encontrar un recurso se reduzca notablemente, pero esto puede convertirse en un problema si el servidor recibe un número elevado de peticiones, por lo que la escalabilidad no es el punto fuerte de esta arquitectura. Asimismo, si el servidor sufre una caída, toda la red se verá comprometida.

Dos claros ejemplos de sistemas P2P centralizados son BOINC [8] y Napster [9].

3.2.2.2 Redes descentralizadas

En las redes P2P descentralizadas [10] también denominadas “puras” no se cuenta con la presencia de un nodo central, sino que cada nodo tiene el mismo rol y capacidades que el resto. La idea base para entender el funcionamiento de estos sistemas es que cada nodo actúa al mismo tiempo como cliente y como servidor.
La característica más destacable de estos sistemas es la alta escalabilidad. Sin embargo, el primer problema a la hora de implementar una arquitectura de este tipo es el de localizar los nodos con los recursos o servicios solicitados, ya que ya no se tiene un servidor central que proporcione esta información. Existen diferentes soluciones frente a este problema que se detallarán más adelante.

Algunos ejemplos de sistemas que hacen uso de esta arquitectura son Freenet [11]y Kademlia [12].

3.2.2.3 Redes híbridas

Para aprovechar las ventajas tanto de un tipo de arquitectura como de otra, en ciertas ocasiones se opta por unificar los dos tipos de arquitecturas explicados anteriormente [13]. El objetivo principal es hacer uso del rápido acceso a los recursos en una red centralizada pero manteniendo una alta escalabilidad como se tiene en una red descentralizada.

No existen nodos específicos que actúen como servidores centrales, sin embargo existen nodos con mayor peso que otros, llamados *super-peers*, los cuales son usados como servidores por otros.

Aunque se haga uso de servidores para localizar recursos y servicios, también se emplean técnicas de búsqueda descentralizadas. De esta manera, se obtiene una red equilibrada que aprovecha los beneficios tanto de una arquitectura como de otra. Como ejemplos de redes P2P híbridas se tienen BitTorrent [14] o eDonkey [15].

3.2.3 Clasificación según la estructura

El principal problema al que se enfrenta el diseño de un sistema P2P es el encaminamiento de mensajes. Existen múltiples maneras de llevar a cabo esta tarea, sin embargo, la cuestión de cómo hacerlo reside en qué información debe contener cada nodo.

El concepto original consiste en que un nodo envía una petición a sus vecinos solicitando un recurso, si algún vecino posee dicho recurso se acabó el encaminamiento y lo devuelve. Pero si un vecino no tiene el recurso, éste a su vez reenvía la petición a sus vecinos. Y así sucesivamente hasta encontrar el recurso. No obstante, en redes centralizadas como Napster [9] este encaminamiento de mensajes y peticiones se ve simplificado pues el nodo central posee la información sobre qué nodo tiene cada recurso.

En la actualidad existen tres soluciones diferentes al problema del encaminamiento basadas en la estructura de la red y en la información almacenada en cada nodo.

3.2.3.1 Redes no estructuradas

Este tipo de redes [16] fueron las primeras que se construyeron, y no dependen de una topología fija, sino que ésta depende de los nodos actuales en el sistema.

La característica principal es que cada nodo almacena él mismo su propio contenido y guarda una lista de enlaces a sus nodos vecinos. Cuando un nodo quiere conectarse a la red le basta con conocer a un nodo y recibir de él su lista de nodos vecinos. Este sistema tiene ventajas como bajo coste de mantenimiento y autonomía a nivel de nodo.

El problema surge con el encaminamiento de mensajes. Si un nodo envía una petición solicitando un recurso, este mensaje se propaga por toda la red empezando por sus vecinos, porque no conoce la ubicación de dicho recurso. De este modo el tráfico en las redes desestructuradas es tan elevado que aparece un fenómeno llamado inundación. Para evitar los problemas que pueda acarrear la inundación se establece un tiempo de vida (Time-to-Live, TTL) a cada petición. Si se alcanza este límite de tiempo antes de que el mensaje llegue a su destino, se cancela la petición, por lo que no se garantiza la entrega de los mensajes.

Todas las implementaciones que existen de redes desestructuradas tienen como objetivo garantizar que los mensajes lleguen a su destino antes de alcanzar el límite definido por su TTL. Los algoritmos empleados en su mayoría son búsqueda en profundidad, búsqueda en anchura o búsquedas basadas en heurísticas.

Algunas de las implementaciones más conocidas de este tipo de redes son Gnutella [17] y FreeNet [11].

3.2.3.2 Redes estructuradas

El problema más destacable de las redes desestructuradas es la baja eficiencia, ya que el tiempo que se tarda en encontrar un recurso puede ser muy elevado o incluso éste puede no ser encontrado aun estando en algún punto de la red.

Con las redes estructuradas [18] este problema se soluciona partiendo de la idea de construir redes con una topología determinada (como por ejemplo en forma de anillo como Pastry [19]). En general, para la mayoría de redes estructuradas, el coste de cada petición está en $O(\log N)$ pasos, donde N es el número de nodos en la red y un paso equivale a un único mensaje entre dos nodos consecutivos. Sin embargo, esta alternativa de red tiene un inconveniente, requiere un alto coste de mantenimiento en consecuencia de la topología de red.

En estos sistemas se toman diferentes medidas cuando un nuevo nodo se incorpora a la red, de tal manera que se establece su ubicación en la red para saber dónde se encuentra. De este modo se puede indexar de algún modo cada recurso en la red de acuerdo a la

topología de esta. Como consecuencia se puede garantizar que si un recurso existe en el sistema, éste siempre será encontrado si es consultado.

Los sistemas P2P estructurados se pueden clasificar en función de cómo sea la capa superior de red, distinguiendo entre sistemas basados en árboles, basados en listas de saltos (skip lists) y basados en tablas de hash distribuidas (DHTs).

Sistemas basados en árboles

Estos sistemas P2P tienen como objetivo indexar la información usando diferentes tipos de árboles. Se pretende crear una jerarquía de nodos en los que los de la capa superior son más estables y eficientes mientras que los de la capa inferior son nodos menos eficientes y de poca confianza. La propuesta considerada como la primera red P2P basada en árboles fue P-Grid [20], usando un árbol binario de prefijos. Otras propuestas usan árboles balanceados o árboles multi-camino entre otras.

Sistemas basados en listas de saltos (skip lists)

Este tipo de redes consisten en una lista doblemente enlazada de ordenamiento múltiple. Se componen de varias listas dispuestas en niveles y los nodos de la red participan en estas listas en cada nivel. La información está ordenada y particionada en rangos de valores en las listas por lo que estos sistemas soportan consultas de rango o de *matching*. Algunos ejemplos de listas de saltos son Skip Graph [21] y Skip Net.

Sistemas basados en tablas de hash distribuidas

Son sistemas que emplean tablas de hash distribuidas [22] (DHT), es decir, el modo en el que se almacenan y consultan los recursos es similar al de una tabla hash. Los nodos y la información se organizan e indexan de tal manera que cada nodo es responsable de un rango de valores y cada unidad de datos es asignada a un valor único obtenido mediante una función de hash uniforme, generalmente SHA-1. Esta estructura de redes es la más usada en la actualidad ya que se consigue gran eficiencia en las consultas. A continuación se explica en profundidad el concepto de DHT para entender mejor el funcionamiento de estos sistemas.

Son tablas hash que almacenan pares clave-valor almacenando los datos de manera distribuida entre los nodos de una red, y permiten consultar el valor de una clave de manera eficiente.

La estructura de una DHT se compone principalmente de un espacio de claves repartido entre los nodos de la red. A su vez, una capa superior (overlay) es la que conecta estos nodos para poder encontrar una clave dentro de este espacio de claves.

El proceso de almacenamiento de un valor en la red es el siguiente. Supongamos una DHT con un espacio de claves de cadenas de 160 bits. Por otro lado tenemos un fichero de nombre F y datos D que queremos almacenar en la DHT. En primer lugar, mediante el algoritmo SHA-1 se genera el hash del F para obtener una clave K de 160 bits. En segundo lugar, se envía a los nodos de la DHT un mensaje put(K, D) que es encaminado hasta el nodo responsable de esa clave K, donde será almacenado el mensaje. Posteriormente, cuando se quiera realizar una consulta de ese fichero almacenado, se

generará otra vez la clave K con el hash del nombre del fichero (SHA-1), se encaminará la petición get(K) por la capa superior (overlay) hasta encontrar el nodo responsable de dicha clave (según el particionamiento del espacio de claves) y el valor asociado a ésta será devuelto.

El particionamiento del espacio de claves es el que permite asignar a cada nodo un conjunto de claves para repartir la información a almacenar entre ellos. Existen varios algoritmos para determinar qué claves posee cada nodo. Los más conocidos son *Consistent Hashing* y *Rendezvouz Hashing.*

Por otro lado, el encargado de gestionar el encaminamiento de los mensajes es la capa superior (overlay), formada por los enlaces entre los nodos. Para cada clave K, existe un nodo que la posee o tiene un enlace a un vecino con un ID "cercano" a K (según el algoritmo de particionamiento de espacio de claves). Haciendo uso de algoritmos voraces se puede obtener una forma eficiente de encontrar el propietario de cada clave. En general, el coste del encaminamiento es de *O(log N)* pasos, donde N es el número de nodos en la DHT. Puede variar en función de la topología y del algoritmo de encaminamiento.

Entre numerosas implementaciones de redes P2P sobre DHT encontramos Chord [23], Kademlia [12], Pastry [19] y TomP2P [24].

3.2.3.3 Redes híbridas

Las redes desestructuradas tienen la ventaja de contar con autonomía a nivel de nodo a la hora de localizar vecinos y almacenar recursos, pero tienen como inconveniente el ineficiente encaminamiento basado en inundación. Por el contrario las redes estructuradas necesitan un alto coste en mantenimiento pero son capaces de encaminar mensajes de manera eficiente. Por eso actualmente se intentan combinar características de ambos tipos de redes para sacar partido de sus ventajas.

La idea básica en las redes híbridas [25] es la de crear una jerarquía de nodos. Existen unos nodos denominados supernodos que forman entre ellos una subred P2P. Cada uno del resto de nodos (nodos comunes o clientes) pertenece a un supernodo y no se conecta con ningún otro nodo que no pertenezca al supernodo con el que está conectado.

El proceso de encaminamiento sigue una serie de pasos: En primer lugar, el nodo cliente que solicita un recurso envía una petición al supernodo al que pertenece. Éste busca en su base de datos cuál de sus clientes tiene el recurso pedido o qué supernodo puede proporcionarlo. Por último, cuando la petición llega al supernodo con el *peer* que contiene la respuesta, este supernodo devuelve la IP de ese *peer* al nodo solicitante. Tras esto, los dos nodos realizan su comunicación.

Como ejemplos de redes híbridas encontramos Edutella [26] y BestPeer [27].

3.2.4 Seguridad en las redes P2P

En este apartado se pueden distinguir dos aspectos: la seguridad del equipo sobre el que corre la aplicación P2P frente a las amenazas de la red , y la seguridad relacionada con la integridad y privacidad de los datos manejados por estas aplicaciones.

3.2.4.1 Seguridad en los equipos

La seguridad es un tema delicado cuando hablamos de redes P2P, ya que si partimos de la premisa de que en este tipo de redes, los nodos son los propios usuarios, la seguridad de estos se ve seriamente afectada. Ya no solo por posible software malicioso, ya que esto se puede solucionar de manera relativamente sencilla, analizando todos los archivos descargados con un antivirus. El problema principal es el acceso que se ofrece a través del puerto que utiliza la herramienta, al abrir los puertos estamos ofreciendo una puerta abierta a nuestro ordenador a todos los usuarios de dicha aplicación.
Se han realizado pruebas sobre programas P2P y se ha conseguido demostrar que por muy seguro que sea el programa, con solo unas cuantas herramientas se pueden acceder a nuestros datos, como puede ser los archivos que estamos descargando, nuestra IP, nuestra ubicación, nuestro número de teléfono y en definitiva toda la información que contenga nuestro ordenador.

Lo más preocupante del tema es, que si ni siquiera hemos configurado el acceso a determinadas carpetas, con una búsqueda en el programa se puede acceder a documentos de carácter sensible (CV, claves, datos financieros y un sinfín de casos más) que por desconocimiento estamos compartiendo.

Además de todo lo expuesto anteriormente, hay riesgos de otras amenazas como ataques por denegación de servicio, todo esto se ve favorecido por la propia estructura de estas redes.

Además en ocasiones, no es suficiente realizar la desconexión de la red ya que en muchos casos las conexiones maliciosas que se abrieron siguen estando activas después de desconectarse.

Para mantener un nivel de seguridad óptimo al usar aplicaciones de este tipo se recomienda:

- Configurar la carpeta en la cual se almacenan los archivos y usarla exclusivamente para ello.
- No usar los servicios de mensajería que estas aplicaciones suelen ofrecer. A través de ellos es sencillo abrir un canal de acceso a un ordenador.
- No usar datos reales en la cuenta de la aplicación.
- Usar herramientas de cifrado de datos.
- Analizar todos los archivos descargados en busca de malware y virus.
- Refrescar la IP pública.
- Reiniciar el ordenador una vez que cerremos el programa, ya que aunque se cierre, pueden quedarse canales de acceso abiertos.
- En el caso de tener más dispositivos en red, configurarlos por medio de un firewall para que no sean accesibles para dicha aplicación.
- Uso de la aplicaciones como por ejemplo PeerGuardian [28], para proteger al usuario, filtrando las conexiones de aquellos usuarios peligrosos.

3.2.4.2 Seguridad de los datos manejados

Para garantizar que los datos manejados entre los usuarios de las aplicaciones P2P son los auténticos al llegar desde un origen hasta un destino así como que el acceso a estos se realiza de forma autorizada existen diversos mecanismos: criptografía para la privacidad de la información, y firma digital y certificados digitales para los procesos de autenticación.

3.2.4.2.1 Métodos de criptografía

Entre los diferentes tipos de criptografía destacan: criptografía simétrica o de clave secreta, y criptografía asimétrica o de clave pública.

Criptografía simétrica o de clave secreta

Este tipo de criptografía utiliza la misma clave para cifrar y descifrar la información, por lo que, todo el que quiera acceder al contenido de la misma deberá conocer la clave con la que se ha encriptado el mensaje.

Este es el punto débil de este sistema de encriptación, ya que interceptar la clave transmitida es realmente sencillo y en consecuencia acceder a la información encriptada.

Otro inconveniente es el tener que aprender y registrar todas las claves de los extremos con las que se tiene una comunicación, de modo que es un problema añadido.

Criptografía asimétrica o de clave pública

Su fundamento es el uso de dos claves, por un lado la clave pública, aquella que se da a conocer a todos aquellos extremos que quieran compartir información privada con el propietario de la clave, es decir, aquella con la que se encripta; y por otro lado la clave privada, aquella de uso exclusivo para el propietario la cual bajo ningún concepto debe ser revelada y es la clave con la que se desencripta la información.

Existen múltiples herramientas para cifrar los datos con este método, un ejemplo es el software libre GnuPG [29].

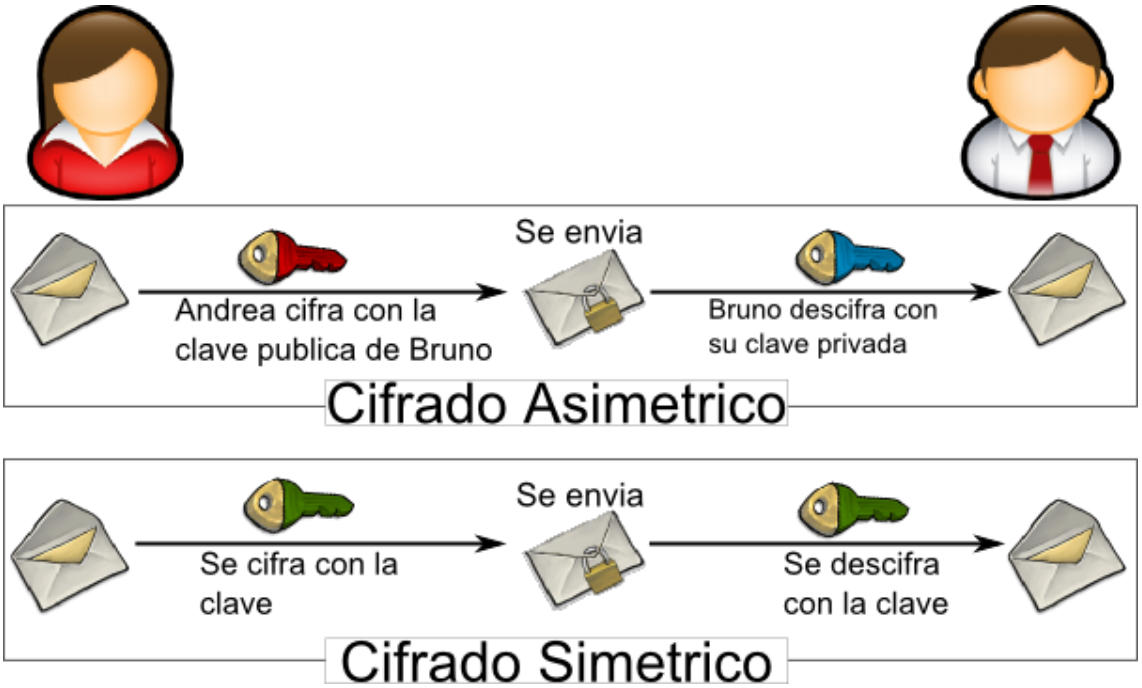

Figura 3.2: Métodos de criptografía [30]

3.2.4.2.2 Métodos de autenticación

Firma digital

La firma digital es una forma de encriptación de información, por la cual se cifran los datos del mensaje y además identifica la identidad de una persona física (la creadora del mensaje).

La firma digital aplica un algoritmo matemático denominado función hash al contenido, una vez realizado esto aplica el algoritmo de firma por el cual se utiliza la clave privada de la persona en cuestión.

La función hash es un algoritmo que calcula un valor numérico de los datos a firmar. Es imposible calcular los datos originales a partir del valor obtenido por la función. Para cada entrada diferente a la función existe una única salida.

El método de encriptación usado en este caso es el de clave pública.

La firma digital se usa para asegurar que el mensaje lo ha generado un usuario en concreto, por ejemplo si el emisor redacta un mensaje y usa su clave privada, el receptor podrá asegurar que ese mensaje lo ha mandado quien dice haberlo hecho, usando la clave pública de la persona que ha emitido el mensaje.

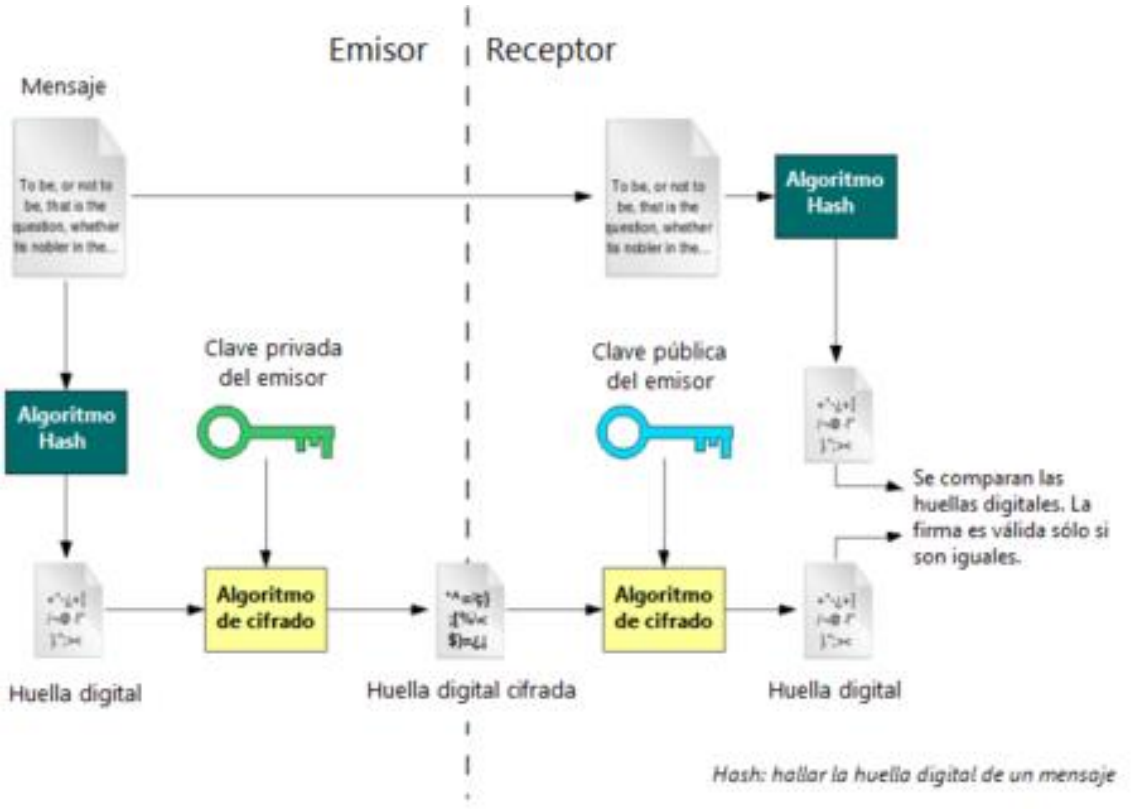

Figura 3.3: Ejemplo firma digital [31]

Certificados digitales

Los certificados digitales son documentos que verifican y validan la relación entre una clave pública y un individuo o entidad. De este modo permiten verificar que una clave pública específica pertenece efectivamente a un individuo determinado. Los certificados evitan el phishing [32] o suplantación de identidad.

El certificado contiene la clave pública y un nombre. Además también contiene la fecha en la que expirará, el nombre de la Autoridad Certificante y un número de serie entre otros campos.

3.2.5 Sistemas de confianza y reputación

Cuando se construye una red en la que no existe una entidad central y las responsabilidades se delegan a los nodos de la misma es necesario controlar la actividad de cada uno de estos para evitar comportamientos no deseados que puedan perjudicar a la red. De este modo se debe implantar un mecanismo para saber en qué nodos confiar a la hora de realizar transacciones en la red.

A la hora de establecer un lazo de confianza entre dos nodos podemos pensar en la confianza del mismo modo que lo haríamos con las personas. Es un concepto que en ciertas ocasiones se basa en opiniones individuales o colectivas. Otras veces se confía en alguien porque ha sido recomendada por una primera persona en la que se confía.

Asimismo la confianza no es algo que sea siempre simétrico, se puede confiar en una persona que no confíe en nosotros.

Todas estas cuestiones han hecho que aparezcan los modelos de confianza, criterios para determinar en qué nodos de la red confiar y cómo.

Los modelos de confianza se clasifican en dos tipos: basados en credenciales y basados en sistemas de reputación. Como se verá a continuación, la confianza no puede basarse sólo en criterios del presente, también es necesario tener en cuenta acciones del pasado.

3.2.5.1 Modelo de confianza basado en credenciales

El funcionamiento de este modelo es sencillo. Un individuo posee unas credenciales, y cuando otro individuo quiere saber si puede confiar en él, le basta con comprobar que sus credenciales cumplen con la política establecida en la red.

El sistema de credenciales [33] más utilizado es el de clave pública/privada. Cuando un agente quiere conectarse al sistema, una clave pública y otra privada le son asignadas, las cuales son usadas para desencriptar y encriptar información respectivamente. La clave pública es almacenada en una entidad de confianza y la privada solo la posee el agente al que va asociada. Cuando se quiere realizar una transacción el agente firma y encripta los datos con su clave privada y posteriormente esta información es desencriptada con la clave pública que posee la entidad de confianza.

Como estas claves se generan una sola vez, se puede garantizar que la información pertenece al agente que dice ser el propietario.

PGP [34] y PolicyMaker son algunos ejemplos de sistemas que usan este modelo de confianza.

3.2.5.2 Modelo de confianza basado en sistemas de reputación

Como se adelantó antes, el problema que tiene el modelo basado en credenciales es que no permite conocer las acciones pasadas de un agente para generar un valor de confianza determinado. Es decir, con conocer sus credenciales no es suficiente para poder confiar en un nodo de una red en la que su comportamiento varía con el tiempo.

Algunos servicios como eBay cuentan con sistemas de reputación [35] para saber en quién se puede confiar, permitiendo a los usuarios que han interaccionado con otro realizar opiniones y valoraciones para futuros usuarios que vayan a realizar transacciones con él.

Existen diversos mecanismo para implementar este modelo, pero aquí se van a tratar algunos sistemas que cubren las ideas principales::

- P2PRep: Las redes P2P gozan de numerosas ventajas que se ven limitadas por sus vulnerabilidades, entre ellas se encuentra el peligro de la aparición de nodos maliciosos que introduzcan contenido de carácter también malicioso (virus,

spyware, etc.). El mecanismo que usa P2PRep consiste en que a la hora de solicitar un recurso a un nodo, el nodo solicitante para conocer la reputación de ese nodo pregunta a los otros nodos que han realizado transacciones previas con él.

El funcionamiento del protocolo básico de este sistema de reputación se define en varias fases:

1. **Búsqueda del recurso**: esta fase consiste en encontrar el recurso utilizando los mecanismo de encaminamiento de mensajes determinado en la red. Al final de esta fase el nodo solicitante tiene una lista de nodos que poseen el recurso solicitado.
2. **Selección del recurso y votación**: una vez recibida la lista de nodos con el recurso deseado. A continuación el nodo solicitante envía un mensaje conocido como *Pool* que pregunta a otros nodos acerca de la reputación de cada nodo que tiene el recurso solicitado.Estos nodos que han sido preguntados responden al nodo solicitante con otro mensaje conocido como *PoolReply* con información sobre la reputación de los nodos que ofrecen el recurso.
3. **Evaluación de la votación**: A continuación se debe comprobar la veracidad o autenticidad del voto pues puede haber nodos maliciosos generando votaciones falsas. Se envía un mensaje *TrueVote* a los nodos votantes y estos responden con otro mensaje *TrueVoteReply*. Ahora que se ha verificado el voto, se elige el nodo con mejor reputación para realizar la transmisión del recurso.
4. **Comprobación del mejor nodo**: Antes de realizarse la descarga se debe llevar a cabo una última comprobación, la autenticidad del nodo proveedor. Primero se envía a éste un mensaje para confirmar su identificador y cuando responde al nodo solicitante, si la respuesta es la correcta, entonces la fase de descarga ya está lista para ejecutarse.
5. **Descarga del recurso**: finalmente se descarga el recurso y el nodo solicitante realiza una evaluación de la experiencia de descarga y actualiza la reputación del proveedor

En el modelo original de P2PRep todas las votaciones que se reciben acerca de un nodo se tratan por igual. Otras implementaciones van más allá y tienen en cuenta aspectos como el nivel de credibilidad de cada nodo en las votaciones, incrementándose o decrementándose en función del éxito final de la transacción sobre la que se realizó el voto.

- XRep: Este modelo de confianza es una continuación de P2PRep. Su protocolo de descarga de recursos sigue teniendo las misma cinco fases pero con la diferencia de que se pregunta por la reputación no solo de los nodos sino también por la de sus recursos. Asimismo cuando se realiza la descarga del recurso se actualiza la reputación de éste a parte de la del nodo proveedor.

Existen muchos otros sistemas de reputación basados en votaciones pero se diferencian básicamente en el modo de realizar dicha evaluación. Tienen en cuenta un histórico de la actividad de cada nodo, valores calculados que expiran para dejar lugar a otros nuevos, etc.

No obstante, en la actualidad, se está dando un paso más allá para mejorar estos modelos y así detectar con mayor eficacia los grupos de nodos maliciosos. A parte de estudiarse el comportamiento individual de cada nodo también se tiene en cuenta su comportamiento con el resto de nodos de la red. Si un agente tiene un buen comportamiento se relacionará con otros agentes de buena reputación y su actividad será abundante, mientras que si existe un agente malo es de esperar que no guarde relación con casi nadie o que si la guarda sea con otros agentes con mala reputación. De este modo se tiene un modelo de confianza más completo. Algunos de los sistemas de reputación que utilizan estos mecanismos son Regret [36] y NodeRanking [37].

3.2.6 Frameworks y alternativas P2P

Para poder empezar con la implementación del banco de tiempo es necesario realizar un estudio sobre qué framework o tecnología P2P utilizar, así como aquellas aplicaciones que han destacado a lo largo de la historia de los sistemas distribuidos. El primer paso ha sido hacer un análisis de cada uno de los frameworks y alternativas más conocidos y relevantes que hemos encontrado. Nuestro propósito es partir de implementaciones de código abierto de los protocolos P2P y sobre todo de tablas de hash distribuidas (DHT) con suficiente documentación. No obstante también hemos investigado sobre otras tecnologías que aunque no vayamos a usar sus posibilidades son interesantes de conocer y estudiar.

3.2.6.1 Open Chord

Open Chord [38] es una implementación de código abierto del protocolo Chord para tablas hash distribuidas (DHT) desarrollada por Distributed and Mobile System Group de la universidad de Bamberg. Está distribuido bajo la licencia GNU GPL [39] y consta de las siguientes características principales:

- Implementación en Java.
- Permite almacenar cualquier objeto Java serializable en una tabla de hash distribuida (DHT)
- Permite crear nuestras propias claves de DHT implementando la interfaz del API de Open Chord.
- Permite réplicas en el almacenamiento en la DHT.
- Permite crear una DHT en una JVM (Java Virtual Machine) con el objetivo de realizar pruebas..

Chord [23] es un protocolo que implementa tablas de hash distribuidas (DHT Chord) y está pensado para funcionar en redes descentralizadas y sin nodos privilegiados. Es uno de los principales protocolos DHT junto con CAN [40] o Pastry [19], entre otros. En una red Chord cada nodo tiene asignado un identificador de *m* bits y se usa una topología en forma de anillo de 2^m nodos (como máximo) en la que cada uno almacena una tabla de nodos vecinos o sucesores. Esta tabla denominada *Finger Table* contiene información sobre qué nodo es el más cercano a una clave dada, y así el proceso de búsqueda se vuelve más eficiente. Asimismo, cuando un nuevo nodo se incorpora a la red, estas tablas deben actualizarse correctamente en toda la red.

3.2.6.2 TomP2P

TomP2P [24] es una implementación para tablas de hash distribuidas avanzadas en las que una clave sirve para almacenar más de un valor. La primera versión fue desarrollada por Thomas Bocek en 2004. Ofrece una interfaz en Java para implementar aplicaciones P2P que se basen en DHTs con operaciones expuestas al programador como get o put (insertar y buscar en la DHT). Está distribuido bajo la licencia Apache [41]. Entre otras, incluye las siguientes características principales:

- Implementación en Java NIO (Non-blocking IO).
- Operaciones típicas sobre la DHT: put y get.
- Permite extender las funcionalidades de la DHT a elección del programador.
- Permite replicamiento del almacenamiento en la DHT.
- Es compatible tanto con IPv4 como con IPv6.
- Permite dejar a la escucha las operaciones sobre la DHT para no bloquear el programa principal.
- Soporta hasta 2^{160} nodos (utiliza IDs de 160 bits).

3.2.6.3 FreePastry

FreePastry [42] es una implementación de código abierto de Pastry [19], un protocolo para implementar tablas de hash distribuidas al igual que Chord o TomP2P. Pastry permite construir una red descentralizada a través de Internet y ofrece varias capas como Past (sistema de ficheros sobre Pastry) o Scribe (sistema de publicación/subscripción para la gestión de encaminamientos y búsqueda de hosts). Es un proyecto que se empezó en 2001 con investigadores de Microsoft Research y se han ido incorporando otras entidades posteriormente (Rice University, Universidad de Washington y Purdue University). Esta distribuido bajo la licencia BSD [43].

Esta implementación de DHTs tiene características similares a las ofrecidas por Chord y TomP2P:

- Implementación desarrollada en Java.
- Ofrece una interfaz para operaciones tales como inserción y búsqueda de contenido en la DHT. Dicha interfaz puede ser implementada por el programador para extender la funcionalidad.
- Proporciona métodos para dejar las operaciones sobre la DHT a la escucha para no bloquear el programa.
- Proporciona una buena persistencia de los datos en la DHT usando réplicas de estos.
- Permite cachear datos de la red en los nodos para reducir retardos de peticiones.

El protocolo Pastry usa una topología de red en forma de anillo. Cada nodo del anillo posee un identificador (NodeID) de 128 bits y es independiente del resto. Estos nodos son localizados mediante tablas de hojas, tablas de vecinos y tablas de encaminamientos que poseen cada uno de ellos. Es tolerante a fallos (si un nodo abandona la red sus vecinos asumen sus responsabilidades) y es totalmente descentralizado siendo capaz de ofrecer un encaminamiento de mensajes dinámico y rápido. También cabe destacar que la función hash usada es SHA-1 por lo que es un protocolo seguro frente a ataques.

3.2.6.4 FreeNet

FreeNet [11] es un sistema P2P completamente descentralizado cuya principal característica es que proporciona anonimato a la hora de publicar contenido en su red P2P, además de ser fuerte frente a la censura con el objetivo de promover la libertad de expresión. La primera versión apareció a principios del año 2000 junto con tres principios:

- Protección de la privacidad, tanto para los usuarios productores de información como para los consumidores.
- Resistente a la censura.
- Alta disponibilidad de la información y un sistema de confianza a través de la red descentralizada.

Está construido sobre una red no estructurada en la que los datos tienen una clave asociada para ser identificados y consultados en la red, pero no existe una relación entre el contenido y su localización como ocurre en las redes estructuradas.

3.2.6.5 Kademlia

Kademlia [12] es una tabla de hash distribuida (DHT) especificada por Petar Maymounkov and David Mazières en 2002. A diferencia de otras DHT, Kademlia reduce el número de mensajes de configuración que los nodos de la red intercambian entre ellos. Cada nodo tiene asignado un ID en un espacio de claves de 160 bits.

Sin embargo, al igual que otras propuestas de DHT el método para almacenar un dato asociado a una clave consiste en guardar esa información en el nodo de la red cuyo ID sea más cercano a la clave (en función de cierto criterio).

Un ejemplo de red que implementa esta DHT es BitTorrent [14].

3.2.6.6 JXTA

Es una tecnología de código abierto (licencia Apache) creada por Sun Microsystem en el año 2001 [44]. Ofrece un amplio grupo de protocolos para la generación de aplicaciones P2P completamente independiente de la plataforma y el lenguaje de programación (Actualmente las versiones más desarrolladas son en Java, C y C++).

Su principal función es el intercambio de mensajes entre los nodos de la red independientemente de la topología de la misma. Asimismo, permite la comunicación entre dispositivos de diferentes tipos, por ejemplo teléfonos, PCs, servidores, etc.

Actualmente se tienen separados los protocolos de la implementación, la implementación de estos protocolos más conocida se denomina JXSE y está desarrollada en Java.

3.2.6.7 JADE

Es una estructura para desarrollar y ejecutar aplicaciones multi-agentes distribuidas, que se basen en la comunicación p2p [45]. Se basa en el paradigma de agentes. Nos permite distribuir toda la información entre los distintos agentes (que serían los nodos) independientemente del tipo de dispositivo (móvil, fijo…) y su conectividad (cable o inalámbrico). Está distribuido como software libre bajo la licencia LGPL [46].

El funcionamiento de JADE es el siguiente, descubre a otros agentes y se comunica con ellos por medio de mensajes asíncronos, independientemente de que los peers sean conocidos o anónimos (ejemplo: mandar un mensaje a todas las personas que se interesen por la cocina).

3.2.6.8 CAN (Content-Addressable Network)

Es una plataforma descentralizada muy tolerante a fallos y fue una de las primeras propuestas de tablas de hash distribuidas. CAN fue propuesta en 2001 en la Universidad de California por primera vez y desde entonces se ha utilizado para construir diferentes sistemas P2P [40].

La implementación de la DHT usa una topología de red basada en un espacio cartesiano multidimensional. Como otras implementaciones de DHT, CAN ofrece escalabilidad, eficiencia, tolerancia a fallos y carga equilibrada.

3.2.6.9 GNUnet

GNUnet es un framework de código abierto distribuido bajo la licencia GNU GPL. Permite crear plataformas P2P proporcionando seguridad y privacidad sin la necesidad de un servicio centralizado [47].

Todos los nodos de la red actúan individualmente como un router y cifran la señal para que la comunicación sea lo suficientemente segura. De este modo la comunicación entre nodos es anónima, de modo que no sería posible diferenciar el nodo que crea el mensaje del que lo recibe.

Lo interesante de este framework, es que tiene un sistema de recompensa a los peer que ofrecen mejores servicios.
No se recomienda su uso si el objetivo no es el anonimato ya que afecta al ancho de banda.

3.2.6.10 Blockchain

Una cadena de bloques o blockchain es una base de datos distribuida [48]. Como su nombre indica, esta tecnología consiste en cadenas de bloques, pero en las que no hay lugar a modificación una vez que un dato ha sido publicado. Se utilizan sobre todo para

almacenar datos ordenados en el tiempo y que no se desean modificar. Las cadenas de bloques pueden funcionar bajo una infraestructura P2P en la que la carga de trabajo se divide entre todos los nodos, y en cada uno de los nodos de la red existe una copia de la cadena de bloques.

La tarea de mayor importancia es la de confirmar los datos o transacciones antes de que sean almacenados en la red. Para ello se sigue un proceso llamado minería que utiliza en el mayor de los casos el algoritmo de Prueba de Trabajo o *Proof Of Work (POW).*

Un ejemplo de uso de este algoritmo es el que implementa Bitcoin [49]. En Bitcoin existen nodos en la red llamados mineros que participan en la confirmación de los datos antes de crear un nuevo bloque en la cadena. Para ello cada minero realiza una prueba de trabajo que consiste en realizar un gran número de cálculos una y otra vez hasta que llega un momento en el que se obtiene un número arbitrario para construir un nuevo bloque.

Este mecanismo evita comportamientos indeseados en la red ya que requiere un proceso de verificación muy costoso para cada dato que se quiere almacenar. No obstante tiene el inconveniente de depender de un alto coste tanto computacional como energético. Asimismo existen propuestas como *Proof-Of-Useful-Work* para aprovechar ese tiempo de procesamiento antes de llegar al resultado para evitar que se pierda tal cantidad de energía y realizar otras tareas útiles.

Hay muchas aplicaciones de consenso distribuido construidas encima de la tecnología blockchain, y seguramente habrá muchas más, incluyendo aplicaciones que se podrían clasificar como de utilidad social como, por ejemplo, Follow-my-vote [50].

3.2.6.11 Volunteer Computing

A parte de las tecnologías mencionadas previamente existen muchas otras alternativas que merece la pena mencionar. Entre ellas se encuentra Volunteer Computing [51]
Es un concepto de sistemas distribuidos en el que propietarios de un computador donan parte de sus recursos (procesamiento, almacenamiento, etc) para un proyecto en común. Un buen ejemplo de un proyecto basado en este concepto fue Great Internet Mersenne Prime Search en 1996.

4 Especificación del banco de tiempo

4.1 Especificación previa del banco de tiempo

Para poder entender mejor los conceptos que se mencionarán en los próximos capítulos a continuación se hará un resumen del estado en el que quedó la especificación del banco de tiempo en el proyecto previo [1]. A pesar de que un resumen de este tipo hubiera sido muy útil para el lector de la memoria del proyecto previo, no estaba presente. Asimismo, se aprovechará el resumen para corregir varios errores presentes en la especificación del proyecto previo así como para completar esta especificación. Dicha especificación del banco de tiempo se divide en cuatro principales subsistemas:

1. **Subsistema básico o de gestión de usuarios**: funcionalidad destinada al registro y baja de usuarios en el sistema así como al inicio y cierre de sesión.

2. **Subsistema de transacciones**: aquí se encuentran aquellas operaciones destinadas a gestionar el pago de los servicios, es decir, el intercambio de las horas. Está dividido en tres partes:
    - **Gestión de contratación de servicios** (llamado "**Gestión de facturas**" en el proyecto previo): antes de realizarse un servicio a cambio de horas, es necesario acordar cuántas horas serán transferidas y realizar el pago de éstas una vez se haya efectuado el servicio.
    - **Gestión de reputación**: tras la realización de un servicio, ambos usuarios realizan una evaluación entre ellos sobre la experiencia.
    - **Gestión de balance/histórico**: es necesario llevar un registro de las transacciones realizadas y del saldo actual de cada usuario.

3. **Subsistema de servicios**: este subsistema es el encargado de la oferta y demanda de los servicios realizados por los usuarios.
    - **Gestión de servicios**: búsqueda, oferta, cancelación, etc. de servicios.
    - **Gestión de ontología**: la idea por la que existe este módulo es porque en el sistema, los usuarios pueden crear nuevas categorías en las que clasificar sus servicios ofertados.

4. **Subsistema de notificaciones**: envío y lectura de notificaciones entre usuarios.

Para cada uno de estos módulos se definieron los requisitos funcionales y casos de uso. A parte se realizaron diagramas de secuencia pero solo para los dos primeros subsistemas. Asimismo se definió un modelo de dominio del sistema con las entidades que participan.

Como el sistema tiene como objetivo ser construido sobre una red P2P, la información intercambiada y almacenada en ésta debe ser tratada en ficheros separados al tratarse de

un sistema distribuido, algunos de los cuales fueron también especificados en esta primera versión del banco de tiempo. De acuerdo a los subsistemas anteriores, se definieron algunos de los ficheros necesarios con diferentes campos y restricciones para cada uno. También se definió el **protocolo de pago**, en el que un usuario que ha recibido un servicio paga al usuario que se lo ofreció con los créditos acordados.

Ficheros almacenados en la red:

- **Private Profile**: perfil privado de un usuario.
- **Public Profile**: perfil público de un usuario.
- **Quote**: presupuesto para el pago de un futuro servicio.
- **Invoice**: factura proforma para el pago de un futuro servicio.
- **Bill**: factura definitiva para el pago de un futuro servicio.
- **Account Ledger Entry**: este fichero almacena la información de un pago (servicio realizado, horas realizadas, usuarios involucrados, etc.), así como el balance de la cuenta antes y después del servicio. El conjunto de los ficheros de este tipo para cada usuario forman su asiento contable o histórico.
- **Feedback By Me (FBM)**: contiene la valoración que el usuario propietario del fichero realizó hacia otro usuario.
- **Feedback About Me (FAM)**: contiene la valoración que otro usuario realizó al usuario propietario del fichero.

Ficheros almacenados localmente:

- **UUID**: identificador único del usuario.
- **Private Key**: clave privada del usuario.
- **Private Profile Hash**: debido a que el sistema fue diseñado para trabajar sobre una red P2P estructurada sobre una tabla hash distribuida (DHT), es necesario almacenar las claves (los hashes) de los ficheros almacenados en la DHT. De este modo, se almacena el hash del perfil privado del usuario.

Por otro lado, surgió el uso de algunos términos los cuales conviene conocer para futuras referencias:

- ***Creditor*** o **acreedor**: usuario del banco de tiempo que presta un determinado servicio a otro usuario.
- ***Debtor*** o **deudor**: usuario del banco de tiempo que recibe un servicio y debe una cantidad de créditos (tiempo) determinada al usuario *creditor* que se lo ofreció.

De este modo, se revisó esta especificación para corregir y añadir detalles que consideramos que faltaban, con aspiraciones de implementar una primera versión de prueba o al menos una demostración de alguno de los subsistemas. Asimismo, aparte de revisar, buena parte del proyecto estuvo orientado a la definición del subsistema de servicios, incluyendo la gestión de servicios y de ontologías.

4.2 Cambios en el análisis

4.2.1 Modelo de dominio

El modelo de dominio define el conjunto de entidades y objetos que participan en un sistema así como las relaciones que existen entre éstos. En el modelo de dominio del banco de tiempo presentado en esta sección (ver Figura 4.1), la clase *Transaction* representa la prestación de un servicio por horas, seguido por su facturación y su pago. La clase central de este modelo es la clase *AccountLedgerEntry*, cada instancia de la cual representa una anotación en un libro de contabilidad, donde el libro que agrupa estas anotaciones se representa por la clase *AccountLedger*. Cada usuario del sistema tiene su propia instancia de *AccountLedger*, es decir, su propio libro de contabilidad. En cada transacción, se crea una anotación en el *AccountLedger* de cada uno de los participantes, es decir, se utiliza la contabilidad por "partida doble" en el que cada asiento contable se compone de dos tipos de anotaciones, uno al debe (en el libro de contabilidad de cada uno de los consumidores del servicio) y otro al haber (en el libro de contabilidad de cada uno de los proveedores del servicio). En el caso habitual, cada transacción implica a un sólo proveedor de servicio y un sólo consumidor de servicio (por el contrario, un ejemplo con relación m-n sería una clase de yoga con dos profesores).

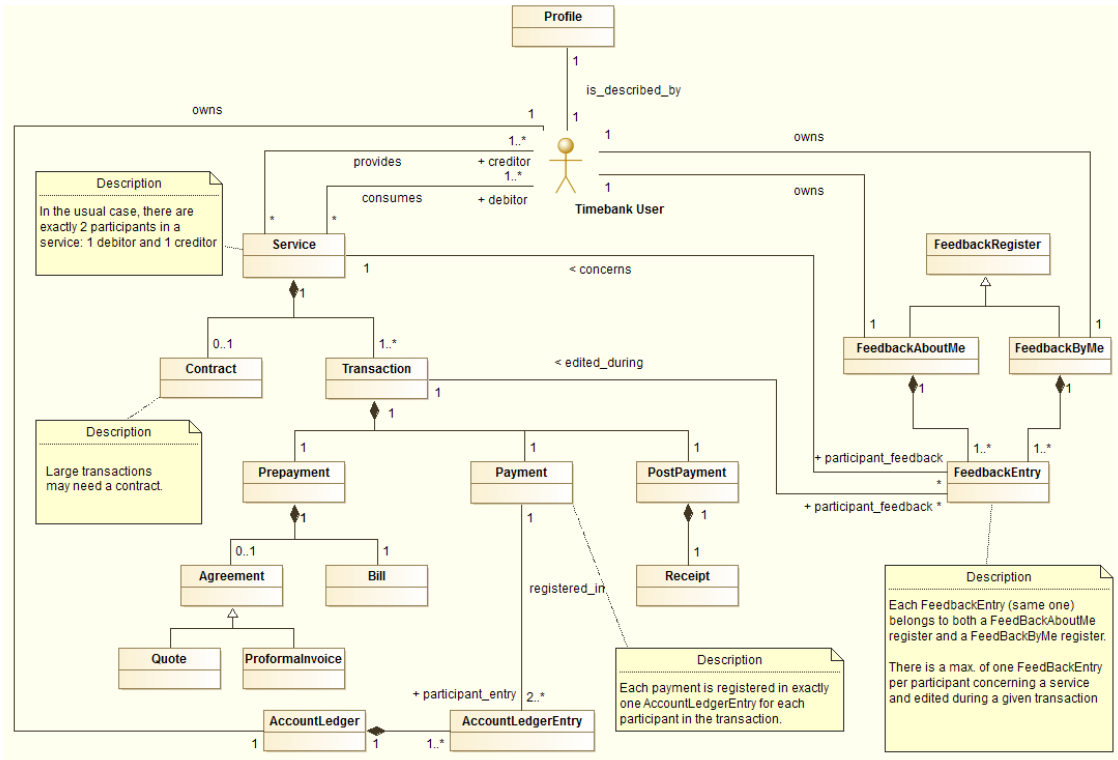

Figura 4.1: Modelo de dominio (diagrama corregido respecto al proyecto previo)

La clase *FeedbackEntry* representa una entrada de realimentación proporcionada, opcionalmente, durante una transacción por uno de los consumidores del servicio sobre uno de los proveedores del mismo o vice versa (aunque el caso más normal es realimentación proporcionada por un consumidor sobre un proveedor, el sistema

también permite el contrario). Cada usuario del sistema tiene dos registros de realimentación: uno de entradas proporcionadas por él sobre otros, representado por la clase *FeedbackByMe*, y otro de entradas proporcionadas por otros sobre él, representado por la clase *FeedbackAboutMe*. Por tanto, una misma entrada de realimentación se almacena en dos registros distintos: el registro representado por una instancia de la clase *FeedbackByMe* del comentador y el registro representado por una instancia de la clase *FeedbackAboutMe* del comentado, con el fin de facilitar el acceso por terceros a la realimentación *sobre* un usuario dado y *por* un usuario dado. La duplicación de realimentación entre registros, cada uno de los cuales que pertenece a una de las partes de la transacción, también ayuda a mantener la integridad de datos frente a fallos y frente a comportamiento malévolo.

Respecto al modelo de dominio definido en el proyecto previo sí se han hecho cambios importantes principalmente en toda la parte relacionada con las transacciones que implica un servicio:

- Ahora en la realización de un servicio no tiene por qué participar solo un único *creditor* y un único *debtor*, puede ser una relación de varios a varios.
- Antes de realizarse un pago tras recibir un servicio debe haberse generado una factura (*Bill*) previa obligatoria y opcionalmente un presupuesto (*Quote*) o una factura proforma (*ProformaInvoice*). En el modelo previo esta factura era opcional y además si se hacía debía ir acompañada obligatoriamente de una factura proforma y un presupuesto.
- Ahora tras realizarse un pago se generará un recibo asociado a esta transacción.
- Ahora el feedback que un participante haga puede hacerse a nivel de servicio o a nivel de transacción (un servicio está compuesto por una o más transacciones).

4.2.2 Casos de uso

En la revisión de los casos de uso se han modificado algunos de ellos para tener una visión más detallada de la interacción del usuario con el sistema. A continuación se muestran los diagramas de caso de uso definidos, diagramas corregidos respecto al proyecto previo.

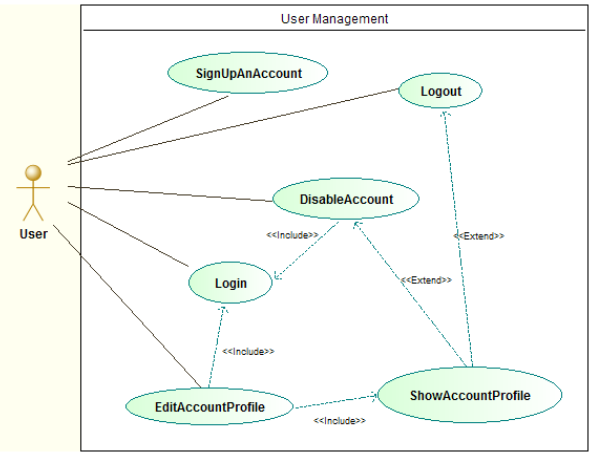

Figura 4.2: Diagrama de casos de uso “Gestión de usuarios”

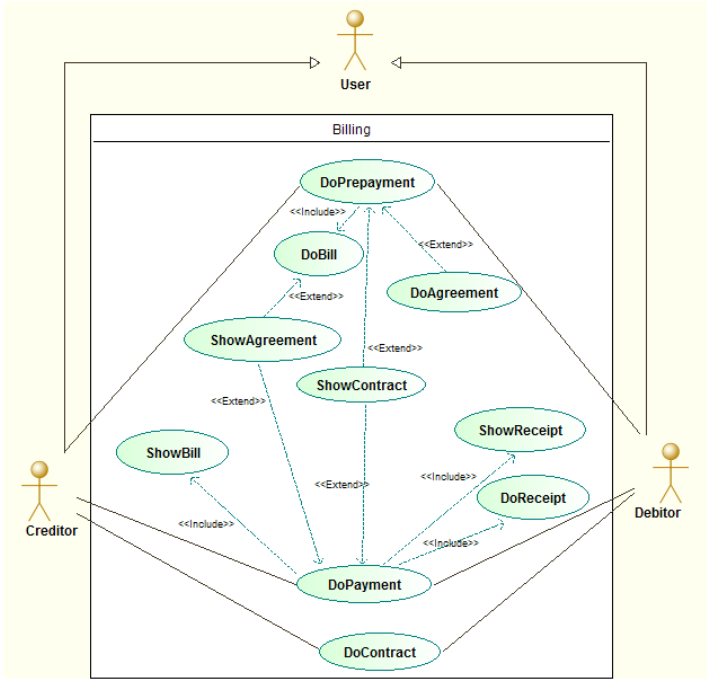

Figura 4.3: Diagrama de casos de uso “Gestión de contratación de servicios” (llamado “Gestión de facturas” en el proyecto previo)

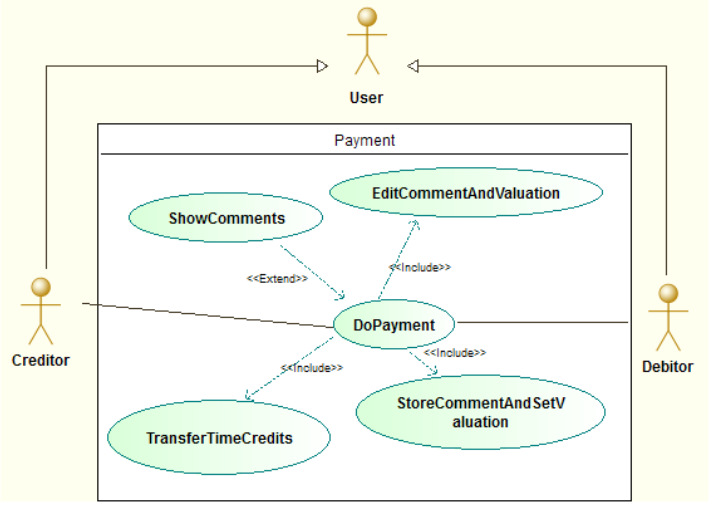

Figura 4.4: Diagrama de casos de uso “Pago”

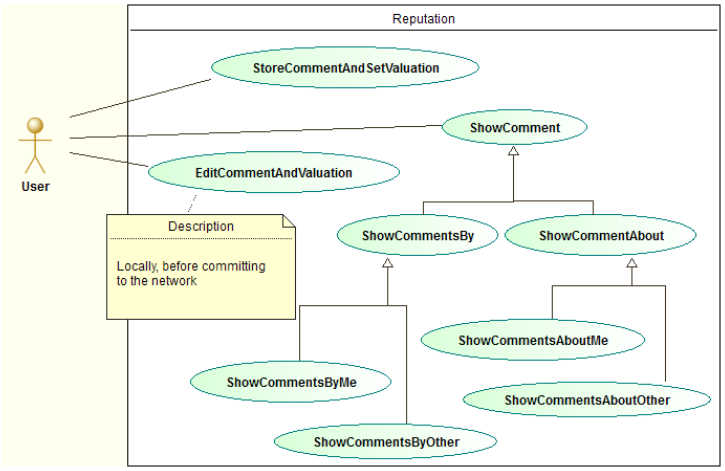

Figura 4.5: Diagrama de casos de uso "Gestión de reputación"

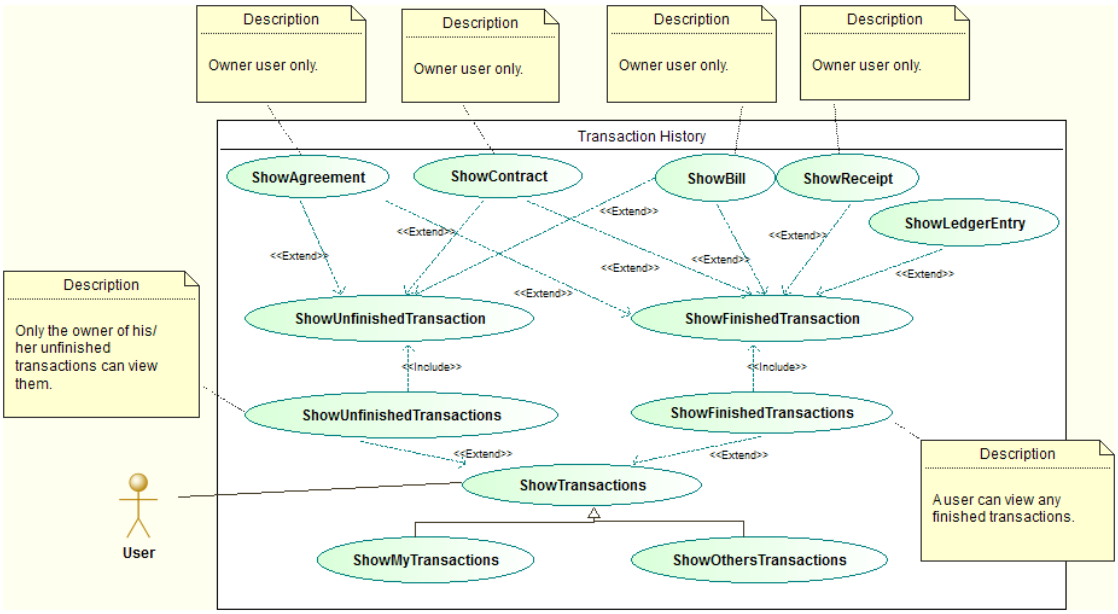

Figura 4.6: Diagrama de casos de uso "Historial de transacciones"

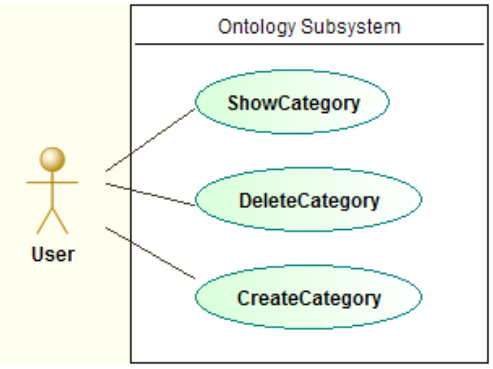

Figura 4.7: Diagrama de casos de uso "Gestión de ontología"

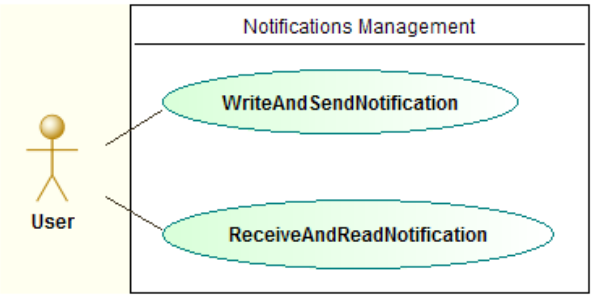

Figura 4.8: Diagrama de casos de uso "Gestión de notificaciones"

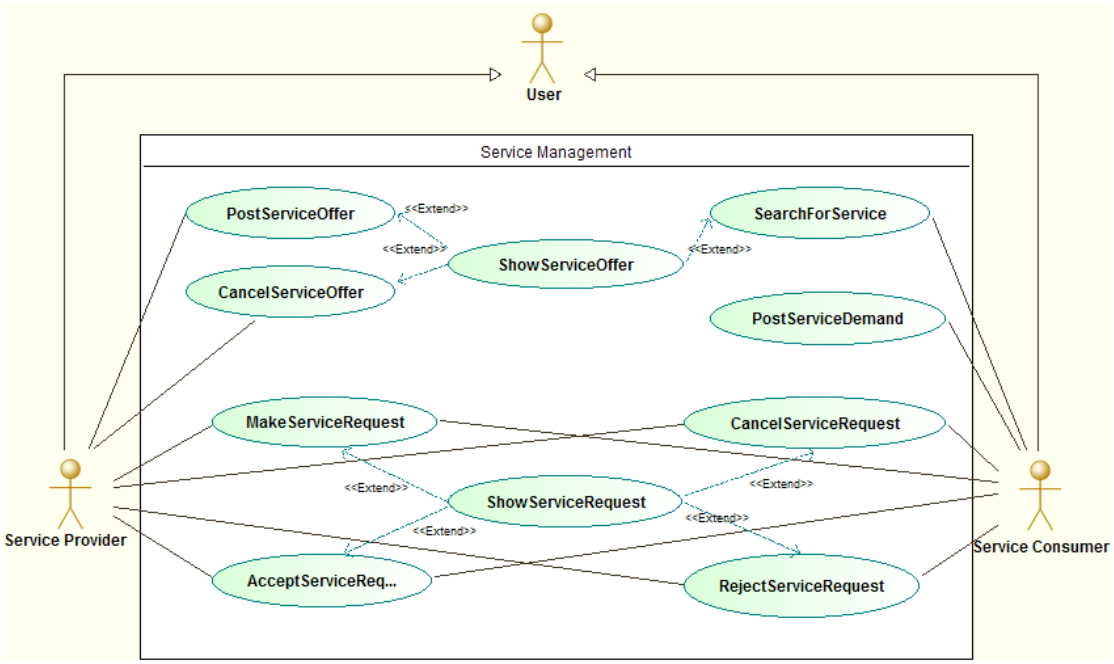

Figura 4.9: Diagrama de casos de uso “Gestión de servicios”

4.3 Cambios en el diseño

4.3.1 Modelo de datos

Además de redefinir el modelo de dominio antiguo se definió también un nuevo modelo de datos basado en él y en los ficheros del sistema descritos anteriormente. En este modelo de datos se trata de plasmar en un diagrama de clases las relaciones de cada fichero con los otros para tener una visión global de los datos que se manejan.

El modelo de datos de las transacciones que se presenta en esta sección (ver Figura 4.10) representa los datos principales utilizados en la implementación de las transacciones del banco de tiempo, es decir, los datos que se almacenarán en la DHT. Como es de esperar, este modelo (propio de la fase de diseño) tiene mucho en común con el modelo de dominio (propio de la fase de análisis), p.ej. cada uno de las partes de una transacción tiene sus propias instancias de las clases *AccountLedgerEntry*, *FeedbackAboutMeEntry*} and *FeedbackByMeEntry*. Sin embargo, hay diferencias importantes entre los dos modelos.

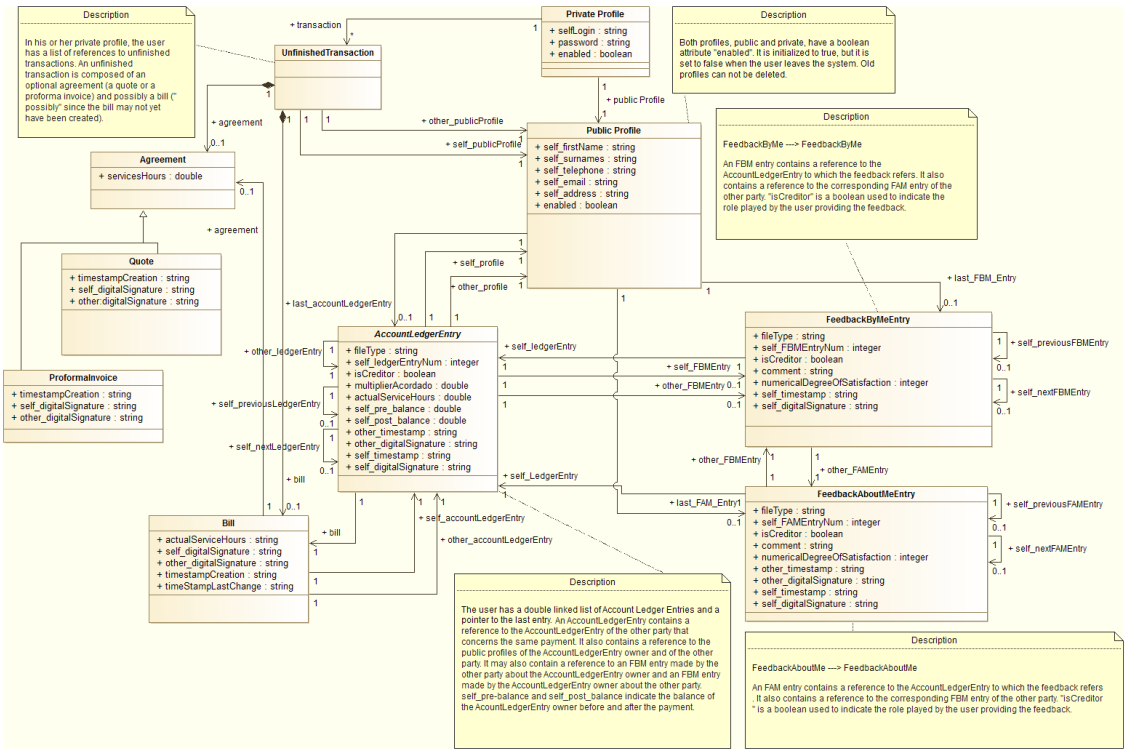

Figura 4.10: Diagrama de clases que especifica el modelo de datos para los pagos[1].

El motivo de la ausencia en el modelo de datos de clases explícitas que representan al libro de contabilidad y los dos registros de realimentación de cada usuario fue el de no tener ficheros mutables en el sistema, con el fin de facilitar la implementación de este diseño sobre una plataforma P2P que solo tiene primitivas de bajo nivel. Por tanto, se tomó la decisión de almacenar la implementación de cada instancia de las clases *AccountLedgerEntry*, *FeedbackAboutMeEntry* y *FeedbackByMeEntry* como un fichero distinto en la DHT. Nótese que, aunque cada una de las dos partes de la transacción tiene sus propias instancias de las clases *AccountLedgerEntry*, *FeedbackAboutMeEntry* y *FeedbackByMeEntry* (también de las clases *PublicProfile* y *PrivateProfile*), las instancias que pertenecen a una de las dos partes de una transacción referencian a las instancias que pertenecen a la otra parte de la transacción. Haciendo la analogía con una base de datos relacional, las clases del modelo de datos se corresponden con las tablas de una esquema de base de datos, una instancia de una clase del modelo de datos (implementado, recordémoslo, como un fichero almacenado en la DHT) se corresponde con una fila de la tablas de la base a la que corresponde su clase, y las referencias entre distintas clases del modelo de datos se corresponden con las claves foráneas del esquema.

Como se ha dicho antes, en el modelo de datos, el libro de contabilidad de cada usuario se modela (implícitamente) como una secuencia de instancias de la clase *AccountLedgerEntry* donde el orden es cronológico. Para permitir recuperar de la DHT la información sobre una anotación cualquiera del libro de contabilidad de un usuario dado, cada instancia de la clase *AccountLedgerEntry* referencia a la instancia anterior y la instancia siguiente en la secuencia (cuando existen), es decir, el libro de contabilidad de cada usuario tiene la forma de una lista doblemente enlazada de instancias de la clase

[1] Nuevo diagrama respecto al proyecto previo.

AccountLedgerEntry. Como punto de partida de esta recuperación, la instancia de la clase *PublicProfile* de cada usuario contiene una referencia hacía la última anotación de su libro de contabilidad. Del mismo modo, los dos registros de realimentación de cada usuario se modelan (implícitamente) como una secuencia de instancias de las clases *FeedbackByMeEntry* y *FeedbackAboutMeEntry* respectivamente, ordenados cronológicamente, ambos registros tienen la forma de una lista doblemente enlazada, y la instancia de la clase *PublicProfile* de cada usuario contiene una referencia hacia la última entrada de cada uno de sus dos registros de realimentación.

Respecto a la implementación de las referencias, cuando una instancia referencia a otra, en el fichero de la DHT que implementa la primera, aparece la clave (valor de hash) del fichero de la DHT que representa la segunda. Finalmente, se puede observar el uso de firmas digitales en el modelo de datos. Cada usuario firma los ficheros que le pertenecen aunque puede que el otro participante en la transacción también haya firmado una versión parcial de la entrada en cuestión. En la siguiente sección se dará más detalle sobre los ficheros efímeros que contienen estas versiones parciales y que se crean durante el protocolo de pago. Las instancias de la clase *Bill* y de la clase *Agreement* pertenecen al proveedor del servicio.

4.3.2 Protocolo de pago

Se han realizado algunos cambios sobre el protocolo de pago definido en el proyecto previo. Se detallarán exactamente los campos que componen cada fichero parcial o efímero y cada fichero completo o persistente, así como el cálculo del hash para su clave en la DHT. Los ficheros y sus campos son los mismos definidos en el proyecto previo [1], en ellos no se han hecho cambios, solo se ha alterado el orden en el que se crean y rellenan los campos durante las distintas fases del pago.

4.3.2.1 Ficheros persistentes

Son los ficheros que tiene que haber por cada usuario tras finalizar el pago, es decir, tiene que haber dos de cada tipo.

- AccountLedgerEntry: Contiene la información relativa al pago.
    - Campos: todos los campos ya definidos en el modelo de datos.
    - Cálculo del hash:
        - ledgerEntryID = concat(accountLedgerString, ledgerEntryNum)
        - ledgerEntry_DHTHash = makeDHTHash(self_UUID, ledgerEntry_ID)

- FBM (Feedback By Me): contiene la valoración que el usuario propietario del fichero realizó hacia otro usuario.
    - Campos: todos los campos ya definidos en el modelo de datos.
    - Cálculo del hash:
        - FBMEntry_ID = concat(FBMString, FBMEntryNum)
        - FBMEntry_DHTHash = makeDHTHash(self_UUID, FBMEntry_ID)

- FAM (Feedback About Me): contiene la valoración que otro usuario realizó al usuario propietario del fichero.
    - Campos: todos los campos ya definidos en el modelo de datos.
    - Cálculo del hash:
        - FAMEntry_ID = concat(FAMString, FAMEntryNum)
        - FAMEntry_DHTHash = makeDHTHash(self_UUID, FAMEntry_ID)

4.3.2.2 Ficheros parciales

Estos ficheros son los mismos que antes pero completos parcialmente ya que el pago está divido en varias fases y en cada una de ellas se añaden ciertos campos a cada uno de los ficheros que tiene que haber.

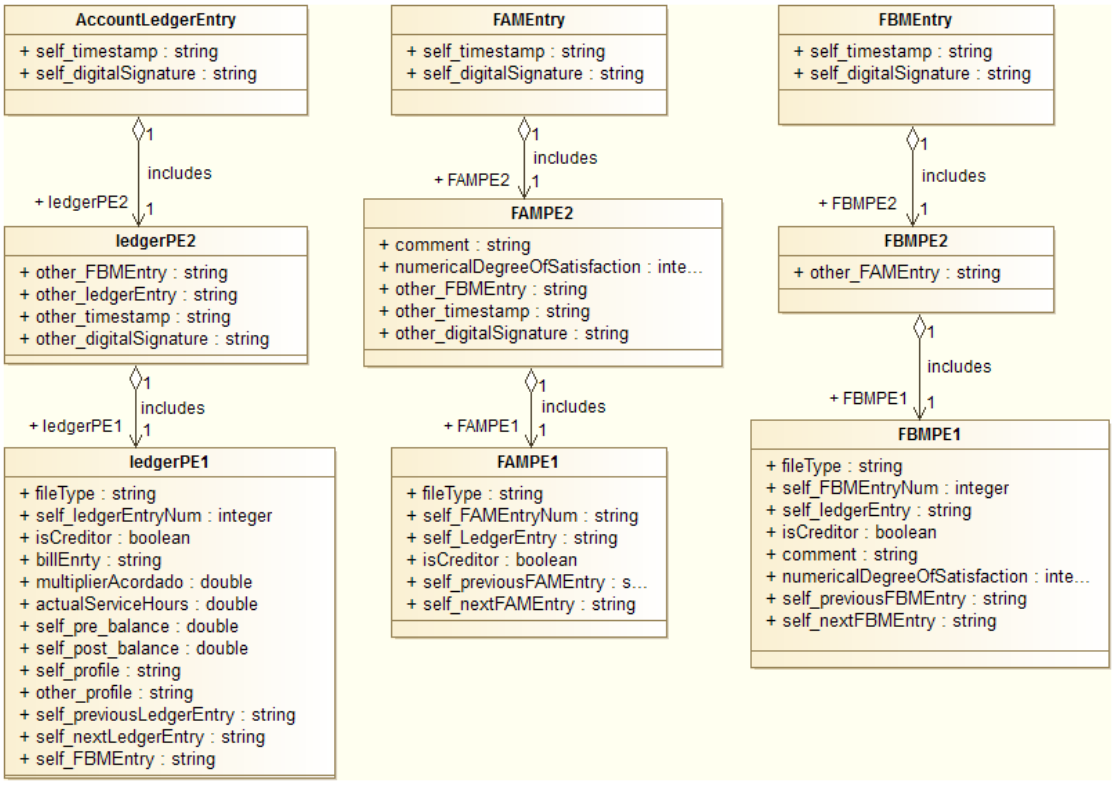

Figura 4.11: Ficheros parciales del protocolo de pago

4.3.2.2.1 AccountLedgerEntry

- ledgerPE1: Primer fichero parcial de un AccountLedgerEntry. Contiene los mismos campos que el fichero persistente excepto los indicados a continuación:
    - Campos pendientes:
        - other_FBMEntry_DHTHash
        - other_ledgerEntry_DHTHash
        - other_timestamp
        - other_digitalSignature
        - timeStamp_creation
        - self_digitalSignature
    - Cálculo del hash:
        - ledgerPE1_DHTHash = makeDHTHash(ledgerPE1 fields)

- ledgerPE2: Segundo fichero parcial de un AccountLedgerEntry. Contiene los mismos campos que el fichero persistente excepto los indicados a continuación:
    - Campos pendientes:
        - self_timestamp
        - self_digitalSignature
    - Cálculo del hash:
        - ledgerPE2_DHTHash = makeDHTHash(ledgerPE2 fields)

4.3.2.2.2 FeedbackAboutMeEntry

- FAMPE1: Primer fichero parcial de un FAM. Contiene los mismos campos que el fichero persistente excepto los indicados a continuación:
    - Campos pendientes:
        - comment
        - numericalDegreeOfSatisfactionWithService
        - other_FBMEntry_DHTHash
        - other_timestamp
        - other_digitalSignature
        - self_timestamp
        - self_digitalSignature
    - Cálculo del hash:
        - FAMPE1_DHTHash = makeDHTHash(FAMPE1 fields)

- FAMPE2: Segundo fichero parcial de un FAM. Contiene los mismos campos que el fichero persistente excepto los indicados a continuación:
    - Campos pendientes:
        - self_timestamp
        - self_digitalSignature
    - Cálculo del hash:
        - FAMPE2_DHTHash = makeDHTHash(FAMPE2 fields)

4.3.2.2.3 FeedbackByMeEntry

- FBMPE1: Primer fichero parcial de un FBM. Contiene los mismos campos que el fichero persistente excepto los indicados a continuación:
    - Campos pendientes:
        - other_FAMEntry_DHTHash
        - self_timeStamp
        - self_digitalSignature
    - Cálculo del hash:
        - FBMPE2_DHTHash = makeDHTHash(FBMPE1 fields)

- FBMPE2: Segundo fichero parcial de un FBM. Contiene los mismos campos que el fichero persistente excepto los indicados a continuación:
    - Campos pendientes:
        - self_timeStamp
        - self_digitalSignature

- Cálculo del hash:
  - FBMPE2_DHTHash = makeDHTHash(FBMPE2 fields)

4.3.2.3 Fases del protocolo

En este punto se detallarán los pasos y fases del protocolo. En cada una de las fases se describen los pasos que se realizan en el lado del acreedor (*Creditor*) y del deudor (*Debtor*):

Fase 1:

1. Debtor:
   a. Se carga la factura (*bill*) correspondiente de la DHT para rellenar los datos del pago.
   b. Con la información de la factura se crean los siguientes ficheros efímeros:
      i. debitorFBMPE1 (FBMPE1)
      ii. CreditorFADebitorPE1 (FAMPE1)
      iii. debitorLedgerPE1 (ledgerPE1)
   c. Se almacenan los 3 ficheros efímeros anteriores en la DHT
   d. Se notifica al Creditor la creación de estos ficheros mediante el envío de pares (ID, Hash)

2. Creditor**:**
   a. Se carga la factura (*bill*) correspondiente de la DHT para rellenar los datos del pago.
   b. Con los pares (ID, Hash) que se han recibido del Debtor se cargan los siguientes ficheros:
      i. debitorFBMPE1
      ii. CreditorFADebitorPE1
      iii. debitorLedgerPE1
   c. Se comprueba que los ficheros anteriores están bien formados.

Fase 2:

1. Creditor**:**
   a. Con la información de la factura cargada previamente se crean los siguientes ficheros efímeros:
      i. creditorFBMPE1 (FBMPE1)
      ii. creditorLedgerPE1 (ledgerPE1)
      iii. DebitorFACreditorPE1 (FAMPE1)
   b. Con los ficheros efímeros recibidos del Debtor se crean los siguientes ficheros efímeros:
      i. CreditorFADebitorPE2: se toman los campos de CreditorFADebitorPE1 y se añaden los siguientes campos:
         1. comment
         2. numericalDegreeOfSatisfactionWithService
         3. other_FBMEntry_DHTHash
         4. other:timeStamp
         5. other_digitalSignature

ii. debitorLedgerPE2: se toman los campos de debitorLedgerPE1 y se añaden los siguientes campos:
    1. other_FBMEntry_DHTHash
    2. other_ledgerEntry_DHTHash
    3. other:timeStamp
    4. other_digitalSignature
iii. debitorFBMPE2: se toman los campos de debitorFBMPE1 y se añade el siguiente campo:
    1. other_FAMEntry_DHTHash

c. Se almacenan los 6 ficheros efímeros anteriores en la DHT
d. Se notifica al Debtor la creación de estos ficheros mediante el envío de pares (ID, Hash).

2. Debtor**:**
    a. Con los pares (ID, Hash) que se han recibido del Creditor se cargan los siguientes ficheros:
        i. creditorFBMPE1
        ii. creditorLedgerPE1
        iii. DebitorFACreditorPE1
        iv. CreditorFADebitorPE2
        v. debitorLedgerPE2
        vi. debitorFBMPE2
    b. Se comprueba que los ficheros anteriores están bien formados

Fase 3:

1. Debtor**:**
    a. Se crean los siguientes ficheros persistentes:
        i. debitorLedger (AccountLedger): se toman los campos de debitorLedgerPE2 y se añaden los siguiente campos:
            1. self_timestamp
            2. self_digitalSignature
        ii. CreditorFADebitor (FAMEntry): se toman los campos de CreditorFADebitorPE2 y se añaden los siguientes campos:
            1. self_timestamp
            2. self_digitalSignature
        iii. debitorFBM (FBMEntry): se toman los campos de debitorFBMPE2 y se añaden los siguientes campos:
            1. self_timestamp
            2. self_digitalSignature
    b. Se almacenan los 3 ficheros persistentes en la DHT
    c. Se crean los siguientes ficheros efímeros:
        i. creditorFBMPE2: se toman los campos de creditorFBMPE1 y se añade el siguiente campo:
            1. other_FAMEntry_DHTHash
        ii. creditorLedgerPE2: se toman los campos de creditorLedgerE1 y se añaden los siguientes campos:
            1. other_FBMEntry_DHTHash
            2. other_ledgerEntry_DHTHash
            3. other_timestamp

4. other_digitalSignature

iii. DebitorFACreditorPE2: se toman los campos de DebitorFACreditorPE1 y se añaden los siguientes campos:

1. Comment
2. numericalDegreeOfSatisfactionWithService
3. other_FBMEntry_DHTHash
4. other_timestamp
5. other_digitalSignature

d. Se almacenan los 3 ficheros efímeros en la DHT
e. Se notifica al Creditor la creación de estos ficheros mediante el envío de pares (ID, Hash)

2. Creditor**:**
   a. Con los pares (ID, Hash) que se han recibido del Creditor se cargan los siguientes ficheros:
      i. creditorFBMPE2
      ii. creditorLedgerPE2
      iii. DebitorFACreditorPE2
   b. Se comprueba que los ficheros anteriores están bien formados.

Fase 4:

1. Creditor**:**
   a. Se crean los siguientes ficheros persistentes:
      i. creditorLedger (AccountLedger): se toman los campos de creditorLedgerPE2 y se añaden los siguiente campos:
         1. self_timestamp
         2. self_digitalSignature
      ii. DebitorFACreditor (FAMEntry): se toman los campos de DebitorFACreditorPE2 y se añaden los siguientes campos:
         1. self_timestamp
         2. self_digitalSignature
      iii. creditorFBM (FBMEntry): se toman los campos de creditorFBMPE2 y se añaden los siguientes campos:
         1. self_timestamp
         2. self_digitalSignature
   b. Se almacenan los 3 ficheros anteriores en la DHT.

4.3.3 Diagramas de secuencia

Los diagramas de secuencia que se modificaron fueron aquellos relacionados con los cambios en el diseño (ver Apéndice). En este caso los que sufrieron cambios fueron los del protocolo de pago descrito en el punto anterior.

4.4 Novedades

Al margen de revisar la especificación funcional y de datos del banco de tiempo se vio que era más prioritario continuar con la definición de uno de los subsistemas principales y más importantes del banco de tiempo, que además era la parte que mejor se podría extrapolar a otras aplicaciones similares. Se está hablando del subsistema de servicios, aquel que consiste en la oferta y demanda de servicios, incluyendo la gestión de una ontología para clasificarlos. A esta definición se le ha dedicado un capítulo propio.

5 Subsistema de servicios

Antes de seguir cabe destacar que en el proyecto previo no se especificó nada del subsistema de servicios que se tratará a continuación.

5.1 Background: Ontologías y taxonomías

Como primer paso previo a la definición del problema que se tratará en este capítulo, conviene introducir el concepto de ontología ya que será la base de lo que se hablará más adelante.

La ontología (del griego *οντος, ontos*, que significa ser, y *λόγος, logos*, que significa estudio, ciencia) es una rama de la metafísica definida como el estudio de la naturaleza y las relaciones del ser [52]. Es decir, se centra en cuestiones sobre qué es lo que existe y cómo se relaciona. Las aplicaciones prácticas más conocidas de la ontología son en el ámbito de las ciencias de la computación y la información.

En el contexto de las ciencias de la computación y la información una ontología es una descripción formal de conceptos en un dominio de discurso, las propiedades de estos (que incluye las relaciones entre ellos) y las restricciones sobre estas propiedades.

La ontología en el campo de la filosofía y la ontología en el campo de la computación han seguido caminos diferentes. Por un lado la filosofía se preocupa por problemas tales como si algo existe o si lo que existe se puede ordenar, y por otro lado en las ciencias de la computación se trata directamente la clasificación de conceptos fijos en base a sus relaciones. Asimismo, a lo largo de los últimos años mientras los filósofos seguían debatiendo maneras de representar y definir el ser, se han ido construyendo ontologías cada vez más robustas. Algunos de los ejemplos más representativos son WordNet [53] y Cyc [54]. Las ontologías en las ciencias de la computación tienen un papel importante en el avance de la Inteligencia Artificial, ya que sirven para representar el conocimiento de una manera inteligible para las máquinas, ya sea sobre un dominio en concreto o sobre todo el espectro de lo que el hombre conoce (finalidad que se pretende alcanzar en la IA) [55].

Sin embargo, estos conceptos clasificados no representan información alguna si no se especifican relaciones entre ellos. Mediante la combinación de relaciones y conceptos se construye una red o estructura que da lugar a la representación final del conocimiento. Actualmente, en las ontologías se definen varios tipos de elementos que conforman la red [56]:

- Individuos: objetos o entidades.
- Clases: tipos de objetos, también denominadas colecciones.
- Atributos: propiedades o características que los objetos pueden tener.
- Relaciones: formas en las que las clases e individuos se relacionan con otros.
- Funciones: estructuras formadas a partir de relaciones y que pueden ser usadas en lugar de un individuo en una declaración.
- Restricciones: descripciones formales en cuanto a lo que debe ser verdadero para que alguna aserción sea aceptada como entrada.

- Reglas: declaraciones con la forma *si-entonces* que describen las inferencias lógicas que se pueden extraer de una aserción.
- Axiomas: aserciones definidas de forma lógica las cuales en conjunto forman la teoría que la ontología describe en su dominio de aplicación.
- Eventos: cambios en las relaciones y atributos.

A partir de lo descrito anteriormente se modelan ontologías, lo que ha dado lugar a la aparición de numerosos lenguajes para poder hacerlo de manera formal, cada uno con sus propias limitaciones y restricciones a la hora de usar en su totalidad o no el grupo de elementos previamente enumerado. Entre los lenguajes más extendidos y usados destacan OWL [57], RDF [58] y CycL [59]. Asimismo, cuando se decide construir una ontología, se debe especificar el dominio del conocimiento que va a representar. Cuanto más específico sea este dominio más incompatible será la ontología. Es decir, aquellos sistemas que hacen uso de ontologías centradas en un solo tema, cuando quieran crecer y expandirse a otros campos, el proceso de adaptabilidad a nuevos dominios será muy costoso.

Una de las relaciones más importantes en una ontología es la relación de *subsumption* que indica que un concepto es más general que otro concepto. Una ontología en la que *subsumption* es la única relación (o única relación importante) se suele llamar una taxonomía.

5.2 Problema a tratar y dónde surge

5.2.1 Origen del problema

La motivación que propició esta idea surgió durante la especificación del banco de tiempo, concretamente en el módulo de oferta y demanda de servicios. En un sistema como éste, es importante dar al consumidor la oportunidad de filtrar la búsqueda de servicios, así como permitir al proveedor ofrecer servicios a aquellos potenciales consumidores.

Supongamos que un usuario del banco de tiempo desea ofrecer sus conocimientos como profesor de matemáticas y decide publicar un servicio para dar clases particulares. Al mismo tiempo, otro usuario necesita encontrar a alguien que ayude a su hijo con las matemáticas. Sin un método ágil y eficiente, el proceso para buscar servicios específicos se vuelve costoso e incluso inviable no solo para el consumidor sino también para el proveedor pues no conseguirá público que consuma sus servicios. Aparece entonces la necesidad de un método de búsqueda selectiva. En la mayoría de aplicaciones de este tipo, la mejor solución acaba siendo (por necesidad) clasificar el producto intercambiado (en este caso servicios) en categorías.

De este modo, volviendo al ejemplo anterior, el servicio publicado entraría dentro de la categoría *matemáticas* o *clases particulares* que a su vez pertenecen a las categorías *ciencias* o *educación*, y éstas a otras todavía más generales. Así cuando el usuario consumidor quiera buscar un servicio de este tipo, navegará desde las categorías más genéricas hasta las más particulares, donde se encuentra el servicio deseado.

Esta organización a modo de taxonomía es muy útil y se adapta muy bien al crecimiento del número de servicios del banco de tiempo: por mucho que se añadan servicios al sistema, estos quedarán clasificados. No obstante, esta organización puede tener limitaciones si es fija, es decir, si existe una estructura de categorías inamovible, en las categorías más particulares se acumulará tal cantidad de servicios que la búsqueda perderá las ventajas que se habían conseguido inicialmente. Por lo tanto surge una nueva necesidad añadida: permitir la evolución de la taxonomía con la adición de nuevas categorías y la eliminación de otras..

Por otro lado, se tiene la dificultad añadida de que se trabaja sobre un sistema distribuido P2P por lo que la solución escogida deberá serlo también. El siguiente problema al que nos enfrentemos una vez elegida la manera de ordenar los servicios es cómo publicarlos, es decir, dónde se almacenará la información de éstos para ser consultada posteriormente. En resumen, el objetivo será construir una solución descentralizada.

5.2.2 Cómo se trata este problema en otras aplicaciones conocidas

La necesidad de clasificar los productos de un sistema es un problema al que se enfrentan numerosas aplicaciones similares al banco de tiempo en cuanto a la idea principal a la que se dedican: oferta y demanda de productos, donde en el banco de tiempo el producto son los servicios. En otros sistemas, aunque esta no sea la utilidad final, también se necesita una organización de este tipo. Por ello, es interesante estudiar cómo abordan esta solución otras aplicaciones.

5.2.2.1 eBay

eBay es uno de los sitios dedicados a la subasta de productos en Internet más extendido en la actualidad. Fue fundado en 1995 y su sede central está situada en San José, California, Estados Unidos. El principal servicio ofrecido es el de las subastas en el que se publica un producto acompañado de un precio de salida y durante el período de tiempo especificado los compradores interesados subirán la oferta. Sin embargo, también se da la oportunidad a los vendedores de anunciar un producto con un precio fijo desde el principio [60].

En esta plataforma de subasta de productos se ofrece un servicio de categorías para que los posibles compradores puedan encontrar fácilmente lo que buscan. Esta estructura es una jerarquía en forma de árbol en la que en cada nodo es una categoría formada por otras categorías hija, y así hasta llegar a categorías hoja con un nivel de particularidad elevado. Y no hay una categoría padre que englobe a todas las demás sino que hay una lista de las temáticas más genéricas siendo cada una un árbol de otras más específicas. Asimismo es importante destacar que los productos clasificados solo se encuentran en los nodos hoja.

A lo largo de los últimos años, ha variado esta lista de categorías generales y los árboles que parten de ellas. Esto es debido a la necesidad de adaptarse a las demandas y ofertas

de cada momento. Por ejemplo, a finales de 2009, eBay contaba con una lista de treinta y seis categorías principales de las cuales una de ellas era una categoría hoja, mientras que actualmente establece una lista de treinta y cuatro genéricas de las cuales ninguna es hoja, y los árboles formados a partir de dicha lista son más ramificados. Asimismo se observa por ejemplo como el avance de las tecnologías también influye, el departamento que antes se denominaba *Cell Phones & PDAs* ahora se llama *Cell Phones & Accessories*, y lo mismo ocurre con *Computers & Networking* que ha pasado a ser *Computers/Tablets & Networking* [61] [62].

Con esto se percibe la obligación por parte del sistema de adaptar la jerarquía de categorías a los cambios en el mercado y en los diferentes aspectos de la sociedad. En consecuencia, con el hecho de tener sólo productos clasificados en las categorías hoja, surgen problemas de reordenación con la aparición de nuevas categorías y con la eliminación de otras más antiguas.

Por otro lado, una característica destacable de eBay es que permite a los usuarios crear su propia "tienda particular" con la función *eBay Store* [63]*s.* En ellas el dueño del *Store* podrá establecer sus propias categorías.

5.2.2.2 Amazon

Amazon es una compañía fundada en 1994 con sede en Seattle, Washington, Estados Unidos, y ofrece servicios en comercio electrónico y en computación en la nube. Ofrece sitios web propios en un gran número de países y posee otras empresas como *Internet Movie Database (IMDb)* o *Twitch* (entre otras) [64].

En Amazon los compradores pueden realizar una búsqueda selectiva de lo que quieren encontrar pues el sistema ofrece una estructura de categorías bajo la que se imponen unas reglas. El número de categorías no es extremadamente elevado, y dentro de cada una existen otras más específicas al igual que en eBay. No solo se filtra por categoría sino que dentro de estas se pueden indicar parámetros de búsqueda (precio, marca, vendedor, etc.).

A la hora de ofrecer un producto para su venta, el vendedor debe cumplir con las políticas que Amazon establece. A parte de los requisitos legales y todo lo que ello implica, se deben respetar y seguir unas reglas sobre la clasificación de los productos. Existen por un lado unas categorías llamadas abiertas (más de veinte) en las cuales el vendedor puede listar libremente lo que va a vender sin permiso de Amazon. Por otro lado, otras categorías no permiten la libre asignación de productos, sino que requieren gozar de un perfil profesional y de la aprobación por parte de Amazon [65]. El objetivo es asegurar la validez, la calidad y el cumplimiento de los estándares en los productos anunciados, y así conseguir la confianza de los compradores en estas categorías.

5.2.2.3 Yahoo!

Yahoo! es una empresa en el sector de Internet y software fundada en 1994 cuya sede está situada en Sunnyvale, Estados Unidos. Tiene como objetivo ofrecer un servicio de

Internet global tanto a negocios como a consumidores. Entre los servicios que proporciona Yahoo! los más destacados son su buscador, el servidor de correo electrónico y un portal de Internet [66].

Dentro del portal se encuentra el famoso servicio conocido como *Yahoo! Respuestas*. Esta plataforma interna consiste en la publicación de preguntas por parte de los usuarios del sitio, las cuales son respondidas por otros usuarios. Del mismo modo que en Amazon o eBay se clasificaban los productos vendidos en categorías, Yahoo! hace lo mismo con las preguntas lanzadas en su web [67].

Pero previamente, ya se aplicó la idea de las taxonomías en su buscador. En la primera mitad de la década de los noventa, no existían los buscadores de webs que hoy en día se tienen. Es por eso que Yahoo! decidió crear entonces un directorio web para centralizar de algún modo el acceso a Internet. Este directorio consistía de nuevo en un conjunto de categorías y subcategorías en las que se iban clasificando las webs de Internet. Fue un tipo de estructura muy popular antes del avance en los motores de búsqueda y se crearon muchos otros a parte del directorio de Yahoo!. En la mayoría de ellos se permitía a los propietarios de las webs ser ellos mismos quienes añadiesen sus sitios al directorio y periódicamente un equipo de editores llevaban a cabo revisiones del directorio. Sin embargo, el directorio web de Yahoo! acabó cerrando sus servicios a finales de 2014.

El fin a esta solución que parecía tan acertada fue causado principalmente por la aparición de Google. Este nuevo motor de búsqueda permitía encontrar no solo páginas web sino contenido deseado en ellas, saltándose el proceso tan largo que requería tener que navegar por toda la jerarquía del directorio hasta encontrar la web buscada.

Pero sin duda el verdadero problema residía en la cantidad de información que pretendía clasificar el directorio. En los inicios de Internet no había un número tan elevado de sitios como lo hay hoy en día por lo que el listado de webs se convirtió en una tarea poco viable [68]. Generaba un alto coste el mantener la clasificación actualizada. En resumen, los sistemas de clasificación masivos no proporcionan una solución cómoda al proceso de búsqueda.

5.2.2.4 DMOZ

DMOZ es un directorio web publicado en varios idiomas creado en 1998. También es conocido como *Open Directory Project* (ODP). El concepto de directorio web es el mismo explicado anteriormente con Yahoo!. La diferencia en este caso es que es abierto, es decir, los enlaces son aportados por cualquier usuario que quiera hacerlo. Asimismo los editores también pueden serlo todas aquellas personas que cumplan ciertos requisitos. DMOZ se proponía como una solución a la dificultad que suponía a los sitios entrar en el directorio de Yahoo!, es decir, los principios de la Wikipedia aplicados a Yahoo! [69].

Pero por la misma razón por la que Yahoo! cerró su directorio, DMOZ terminó por hacer lo mismo en marzo de 2017 [70]. Es un ejemplo más de que la clasificación por categorías no resulta una alternativa viable en el caso de cantidades muy grandes de datos.

5.2.2.5 Otros ejemplos

Existen muchos sistemas con el problema que se desea tratar pero al final la solución que se escoge acaba siendo categorizar los productos. Muchas aplicaciones de venta de productos de segunda mano optan por ir un paso más allá y en vez de incluir sólo categorías, facilitan la búsqueda agrupando productos por ubicación, es decir, al igual que se puede clasificar por temática, se clasifica por lugar de venta. Ejemplos de esto son *Wallapop* [71] o *Milanuncios* [72] (aplicación de compra y venta de productos de segunda mano y servicio de tablón de anuncios respectivamente), donde se clasifican los resultados de la búsqueda por ciudad/provincia aparte de por temática. Otros métodos realizan una clasificación dinámica en función de la popularidad actual de cada producto.

También existen otras aplicaciones destinadas a la oferta y demanda de servicios al igual que el banco de tiempo como *Heygo* [73] solo que en este caso es a cambio de dinero. Este sistema mantiene una clasificación con dos niveles de categorías a la que se añade el filtro de provincia.

En conclusión, la solución al problema en el mayor de los casos es crear una taxonomía no estática que se adapte a cambios en función de la oferta y demanda de los usuarios en cada momento.

5.2.3 Cómo se trata este problema en el contexto de la estandarización

Hasta ahora se ha visto cómo se organizan los elementos en aplicaciones típicas de compra y venta de productos entre otras, pero este problema también se trata en el contexto de la estandarización. Un ejemplo de entidad que lleva a cabo esta labor es WIPO (World Intellectual Property Organization) [74] y un ejemplo de estándar es UNSPSC® (United Nations Standard Products and Services Code®) [75].

WIPO (OMPI en español) es una organización mundial destinada a la gestión de la legislación, solicitudes y demás procedimientos en materia de propiedad intelectual. Para ello cuenta con un sistema llamado Clasificación de Niza [76] para clasificar productos y servicios para la solicitud de marcas. Este sistema consta de 34 clases para los productos y 11 para los servicios. Cada clase representa un tipo de producto o servicio y cada una contiene una lista de términos para definirlos de manera más detallada. En este caso, el nivel de clasificación no es tan profundo si no que es prácticamente plano. Por otro lado el UNSPSC está destinado a la clasificación a nivel mundial de los productos y servicios para su uso en comercio electrónico y cuenta con una jerarquía de cuatro niveles codificada con 8 dígitos decimales, donde cada combinación de estos representa una rama concreta de la taxonomía [77]. Asimismo, también existe un sistema fijo de clasificación arborescente para los tipos de contratos públicos en la Unión Europea [78].

Al igual que en los ejemplos previos las clasificaciones a modo de taxonomía se empleaban para facilitar las búsquedas y publicaciones, estos últimos casos expuestos tienen como fin el mismo objetivo pero en un contexto estandarizado, para que entidades de negocio puedan publicar sus servicios de un modo controlado y los clientes y usuarios encontrarlos.

5.2.4 Conclusiones

Tras haber analizado cómo se aborda el problema de la publicación de productos o servicios en múltiples aplicaciones y entidades conocidas, se han extraído algunas conclusiones sobre las ventajas e inconvenientes de sus soluciones propuestas.

A simple vista, parece que en todo sistema dedicado a ofrecer publicación y búsqueda de servicios y productos o a gestionar diferentes tipos de objetos, una buena idea es usar un modelo de categorías. De este modo las entidades del sistema estarán clasificadas y accesibles según lo que se busque o anuncie. Sin embargo, como se ha visto en los ejemplos anteriores, este modelo de información no ofrece sólo ventajas, sino que también genera inconvenientes.

En el caso de que la información pertenezca al mismo ámbito o se encuentre en cantidades no masivas, la solución de crear una jerarquía de categorías con varios niveles es una alternativa aceptable pues se tiene una estabilidad buena sobre el estado de la clasificación. Esto ocurría en los inicios del directorio de Yahoo! o DMOZ cuando no existía un número tan elevado de sitios web como en la actualidad. No obstante, en cuanto el volumen de objetos a clasificar aumenta surgen serios problemas:

- Complejidad para mantener la clasificación actualizada con periodicidad.
- Decisión sobre la ubicación de los elementos en la clasificación: solo en categorías hoja o en cualquier nodo de esta.
- Necesidad de crear nuevos niveles en la clasificación así como eliminarlos. Y consecuentemente:
    - Qué hacer con aquellos elementos que pertenecían a una categoría que ya no existe.
    - Qué criterio escoger para reordenar los elementos conforme a las nuevas categorías añadidas.
        - Si solo existen elementos en las hojas, el sistema está obligado a establecer una solución para decidir dónde reubicarlos.

5.2.5 Caso de uso

Un ejercicio interesante para entender cómo funcionan estos sistemas de taxonomía es realizar un caso de uso de cómo publicar y buscar un producto o servicio en un sistema del tipo de los mencionados anteriormente. Para este caso de uso se supone un banco de tiempo, donde existen dos tipos de usuarios: el ofertante y el demandante.

Anuncio de un servicio:

1. Supongamos un usuario ofertante desea ofrecer servicios como profesor particular de matemáticas para alumnos de secundaria.
2. Suponemos que el usuario se ha dado de alta previamente en el sistema. Tras ello usa la opción "Publicar servicio" y una vez rellenos los campos acerca del servicio (título, descripción, etc) el siguiente paso es clasificarlo en una de las categorías del sistema para facilitar su futura búsqueda.
3. Por lo general, en estos sistemas se permite que un servicio pertenezca a varias categorías. El usuario elige aquellas que considera más adecuada de entre todas las posibles. La secuencia de categorías (de ejemplo) elegida en la jerarquía será: *Educación > Secundaria > Ciencias > Matemáticas.*
4. Una vez finalizado el proceso de clasificación, el servicio que el usuario desea ofrecer ya está a la disposición para que otro usuario que busque un servicio de este tipo pueda encontrarlo fácilmente.

Búsqueda de un servicio:

1. Supongamos ahora el proceso inverso al anterior. Un usuario desea encontrar servicios relacionados con clases de matemáticas para alumnos de secundaria.
2. El primer paso al entrar en el sistema es acceder a la lista de categorías de más alto nivel. Una lista de éstas (simplificada) puede ser la siguiente:
    - Salud
    - Transporte
    - Reparaciones
    - Educación

    Intuitivamente el usuario elegirá la categoría "Educación". Tras esto, se mostrará una sub-lista de categorías dentro de ésta:
    - Primaria
    - Bachillerato
    - Universidad
    - Secundaria

    Se elige la subcategoría "Secundaria" y otra vez se despliega otra lista:
    - Idiomas
    - Literatura
    - Ciencias

    Y una vez seleccionada "Ciencias" se mostrará de nuevo otra lista más específica todavía:
    - Física
    - Química
    - Matemáticas

    Tras haber navegado a lo largo de la jerarquía, el usuario selecciona "Matemáticas" y se encuentra finalmente con un listado de servicios de lo que buscaba.

5.2.6 Problema en términos de ontología

Volviendo al concepto que se introdujo al inicio de este capítulo, podemos entender los sistemas anteriores de clasificación como casos particulares de ontologías. En este caso, el conocimiento que pretenden representar es el conjunto de departamentos de una tienda, en un sistema de comercio electrónico, o el conjunto de posibles categorías permitidas en sistemas de estandarización, por ejemplo.

Las clases o colecciones que conforman la ontología son las categorías, y los individuos o instancias son los productos o servicios clasificados dentro de ellas. Las relaciones que existen entre las clases de estas ontologías son relaciones de parentesco, es decir, las categorías más particulares son hijas de las más generales.

De este modo el concepto de sistema de clasificación distribuida de objetos se puede generalizar a **ontología descentralizada** porque se trabaja con un sistema distribuido.

5.2.7 Tipo de datos más adecuado

Ahora que ya se ha definido una primera aproximación de lo que se pretende construir, el siguiente paso es especificar la estructura de datos que mejor se ajusta.

Las consideraciones y planteamientos a tener en cuenta para ello son los siguientes:.

- ¿Puede haber instancias en todas las clases de la ontología o solo en las clases hoja?
- ¿Una clase puede ser hija de solo una clase o de varias a la vez?
- Las clases guardan relación de parentesco (herencia) en una sola dirección (sin ciclos).

Las clases de la ontología guardan relación entre sí como si fueran una jerarquía. Al principio se pensó en utilizar un árbol como estructura de datos, pero se pensó que la propiedad de multi-parentesco podría ser útil, por lo que la siguiente alternativa que se estudió fue usar un poliárbol.

Un poliárbol es un tipo de grafo acíclico dirigido, cuyo grafo no dirigido subyacente es un árbol [79]. Esta idea de árbol tiene la característica de permitir la presencia de varias raíces, algo que se adapta muy bien a la idea de querer tener nodos con varios padres. No obstante, tiene limitaciones en lo que se busca puesto que en algunas situaciones dejaría de ser un poliárbol. Por ejemplo, supongamos que en un momento dado de la ontología en un subconjunto de ésta se tienen tres conceptos: A, B, C y D. D es padre de A y B, y estos a su vez son nodos hermanos en el árbol y ambos son padre de C. En este caso de ejemplo se pierde la propiedad característica de los poliárboles ya que al sustituir las aristas dirigidas por aristas no dirigidas el grafo resultante sería conexo pero no acíclico.

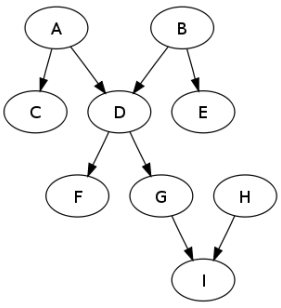

Figura 5.1: Ejemplo de poliárbol

Es por esta razón por la que los poliárboles a simple vista parecían una solución aceptable para la ontología, pero debido a su nivel de particularidad los convierte en una estructura limitada.

Tras esto se investigó el uso del DAG (*Directed Acyclic Graph*), un caso más general que engloba a los poliárboles. La característica principal de este tipo de estructuras, que aunque se ha hablado previamente de árboles esta ya no es uno, es que no permite la existencia de ciclos, es decir, si se parte de un vértice ***v*** del grafo y se sigue una secuencia de vértices conectados sea cual sea, se garantiza que no se volverá a pasar por ***v***.

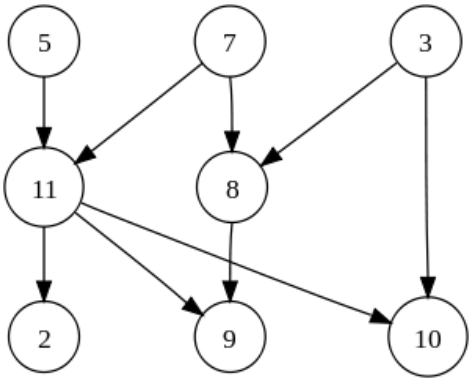

Figura 5.2: Ejemplo de DAG

Este tipo de estructuras son idóneas para construir ontologías, y se suele partir de una sola raíz (Rooted DAG) aunque dependiendo de la situación puede haber más de una.

Con la elección del DAG se solucionan dos de las premisas que se deben cumplir (multi-parentesco y sin ciclos). La otra premisa restante (instancias solo en clases hoja o en cualquiera) es en realidad una decisión de diseño y no una característica de la estructura de datos elegida, ya que el DAG se adaptará a ella se elija una opción u otra.

En nuestra ontología descentralizada se ha optado por permitir la instanciación en cualquier clase aunque no sea hoja. La razón por la que se ha decidido optar por esto es simplemente por viabilidad a la hora de especificar la solución final. Se ha visto en ejemplos como eBay (instancias sólo en hojas) que es necesario establecer reglas y pautas a la hora de eliminar categorías de la clasificación pues no se pueden quedar objetos desclasificados. Lidiar con estos problemas en una entidad centralizada es factible, sin embargo, en un sistema descentralizado sin una única autoridad que tome decisiones se convierte en un problema de diseño mayor.

En un principio se pensó que la solución más sencilla de implementar sería un árbol, pero posteriormente se decidió volver a la idea original de usar un DAG. Más adelante cuando se hable de la especificación de la solución final se justificará esta elección. En resumen, el tipo de datos escogido es un **DAG** y puede haber instancias en cualquier clase de la ontología, no sólo en las hojas.

5.3 Gestión de ontologías dinámicas

Hace ya muchos años que se ha visto la necesidad de usar ontologías que pueden cambiar, y se ha trabajado en la gestión de estos cambios. La gestión de los cambios de una ontología incluye:

- Evolución de la ontología.
- Versionado de la ontología.
- Integración en la ontología (dominios distintos).
- Combinación (*merge*) de ontologías (mismo dominio). Una parte importante del *merge* de ontologías es el *matching* de ontologías [80].

Existen diversas herramientas para la construcción y tratamiento de ontologías, y entre ellas destaca Protégé [81], la cual ha tenido un *plugin* en sus versiones de los últimos años para el *merge* de ontologías [82]. Pero desde la versión cuatro del programa esta funcionalidad ya viene incluida, funcionalidad que requiere menos intervención con el usuario comparada con las versiones previas.

El objetivo de este proceso de investigación es encontrar una alternativa construida en una red distribuida P2P para la gestión de una ontología dinámica, pero es interesante antes analizar qué trabajo previo existe en otras implementaciones tanto descentralizadas como centralizadas.

5.3.1 Repositorio y control centralizado

Esta solución es la más común y usada en la mayoría de aplicaciones. Consiste en la presencia de un repositorio central en el cual se almacena la ontología.

En este tipo de arquitecturas no existen inconvenientes destacables sobre el acceso a la información como puede haber en un sistema distribuido. Aun así surgen algunas dificultades. Como ya se ha hablado antes, el uso de una taxonomía, ontología o clasificación es la solución que parece más acertada para organizar las búsquedas y ofertas. Sin embargo, estas estructuras no pueden ser fijas, con el paso del tiempo deben adaptarse a cambios en la organización. En un repositorio centralizado esta tarea recae sobre un equipo propio del sistema y el coste para mantener la clasificación al día por lo general es elevado.

Este problema es al que se enfrenta entre muchas otras entidades, eBay. eBay cuenta con un sistema de clasificación en continuo cambio sobre el que se aplican ciertas reglas a la hora de reordenar instancias que se ven afectadas por cambios en la jerarquía. Por ejemplo, cuando una categoría crece, es decir, pasa a tener otras categorías hija, como eBay solo permite a los productos pertenecer a una categoría hoja, estos deben reordenarse de acuerdo a estas nuevas categorías. El proceso seguido consiste en asignar al producto la categoría que mejor se adecue en base a la comparación de palabras clave, y si el vendedor de éste no está de acuerdo, él mismo puede elegir otra más adecuada. Lo mismo ocurre cuando dos categorías pasan a ser una sola o cuando una es eliminada [83].

En resumen, lidiar con este tipo de soluciones no termina de ser cómodo o estar del todo claro para los usuarios productores de una aplicación, por eso, las ventajas que proporciona un repositorio centralizado para los consumidores no evitan la aparición de problemas funcionales o de especificación a la hora de anunciar productos y servicios.

5.3.2 Repositorio central, control descentralizado

Es cada vez más común que la construcción y gestión de ontologías, en particular la gestión de cambios, se haga de forma colaborativa, por ejemplo en WebProtégé [84]. A las ya grandes dificultades de la gestión de cambios de ontologías se añaden las que vienen de la colaboración y, habitualmente en este contexto, del acceso remoto y concurrente. En los últimos años ha habido mucho trabajo en esta área.

5.3.3 Repositorio y control descentralizado

Utilizar un repositorio descentralizado con control también descentralizado es el caso que concierne a este proyecto, pues todavía no existen muchos trabajos previos a este problema [85]. Construir una solución descentralizada (P2P) es el objetivo del proyecto por diversos motivos de los que se han ido hablando. Entre ellos, uno de los más importantes es evitar los costes que supone mantener un repositorio centralizado, y se consigue repartiendo entre los participantes de la red el trabajo de administración. En este caso la administración incluye el sistema de clasificación dinámico del que se hablará en este capítulo.

Son los sistemas menos comunes para implementar la solución que se busca debido a sus grandes dificultades añadidas. El sistema está construido sobre una red distribuida y no existe un almacenamiento central donde reside toda la información, de este modo mantener el estado de la clasificación actualizado en toda la red es un trabajo que no puede ser realizado por una sola entidad, sino que recae en los nodos de la red.

5.4 Introducción a la idea buscada

El objetivo es construir un sistema de oferta y demanda de servicios usando una clasificación como hacen otras entidades de las que ya se ha hablado anteriormente. Pero antes de profundizar en la definición y especificación de la solución parcial o final es necesario introducir cuál es la idea básica que se ha pensado para el funcionamiento global de esta parte de la aplicación.

En primer lugar, respecto a la clasificación de los servicios del banco de tiempo, se parte de una red P2P en la que cada nodo posee una versión de una ontología (no necesariamente iguales entre sí) formada por las categorías de los tipos de servicios que se ofrecen. Para mantener el máximo nivel de semejanza entre la ontología de cada nodo, cada uno hará un multicast periódico en la red con una copia de la suya. De este modo cuando se reciba la ontología de otro *peer*, se realizará una operación que combine (***merge***) ambas versiones con el fin de actualizar la copia local que se tiene, y así hacer que el sistema tienda a un estado estable. Se ha pensado la necesidad de incluir con la aplicación una versión inicial de la ontología con una estructura común para

todos los usuarios y así evitar un largo y difícil proceso de consolidación de la clasificación tras numerosos *merges* entre ellos.

Cada usuario del sistema puede crear nuevas categorías en la ontología así como borrarlas, acto que se verá reflejado en el sistema cuando se propague la nueva copia hacia otros usuarios del modo que se acaba de explicar.

Y en segundo lugar, para poder ofrecer y buscar servicios el proceso es el siguiente:

1. La información de cada servicio ofertado debe almacenarse de tal manera que esté siempre disponible para el propietario y sea éste quien tenga el control sobre ella. Puede estar almacenada sólo en el propio nodo del usuario o en supernodos dependiendo de la red P2P empleada. El usuario ofertante permanecerá a la espera de recibir solicitudes de sus servicios. Asimismo estos servicios locales estarán clasificados en base a la ontología de categorías propia del nodo.

2. Cuando un usuario desea buscar un servicio de una determinada categoría, elige una de ellas en la versión local que tiene de la ontología. Tras esto, se guarda el camino o el subgrafo (puede haber varios caminos) desde la categoría solicitada hasta la raíz del árbol, y se envía como un mensaje de petición al resto de usuarios de la red distribuida (el uso de o bien caminos de categorías o bien subgrafos de categorías para los mensajes de petición es una decisión de diseño). En el momento en que un nodo recibe una solicitud de este tipo, comprueba si el camino o subgrafo de categorías indicado en el mensaje *matchea* con la de algún servicio de los que tiene clasificados en local usando su propia ontología. Si se produce algún *match*, se enviará una respuesta de confirmación al usuario demandante del servicio. Por otro lado, si finalmente se decide que se usarán caminos de categorías y no subgrafos, si la ontología tiene forma de DAG y existen varios caminos desde la raíz hasta el nodo, ¿cómo se decide qué camino escoger?, ¿se haría con intervención del usuario (el buscador del servicio)?

5.5 Análisis

Una vez descrito el objetivo que se pretende especificar, es el momento de estudiar las particularidades de cada aspecto de la solución. Primero se empezará tratando los problemas planteados en la idea básica: funcionamiento de la ontología (operación *merge*, estructura de datos, parámetros, etc.), oferta de servicios y solicitud de servicios. A continuación se hará un estudio sobre la infraestructura de red sobre la que se construirá la aplicación.

5.5.1 Ontología de categorías

Uno de los pilares del correcto funcionamiento del sistema es la ontología de categorías. Esta estructura mantiene una clasificación de las categorías de los servicios que se ofrecen en el banco de tiempo. Se ha visto anteriormente que mantener una clasificación estática no resulta eficaz por lo que esta ontología deberá ser dinámica. Un ejemplo de ontología es el de la Figura 5.3: Ejemplo de ontología.

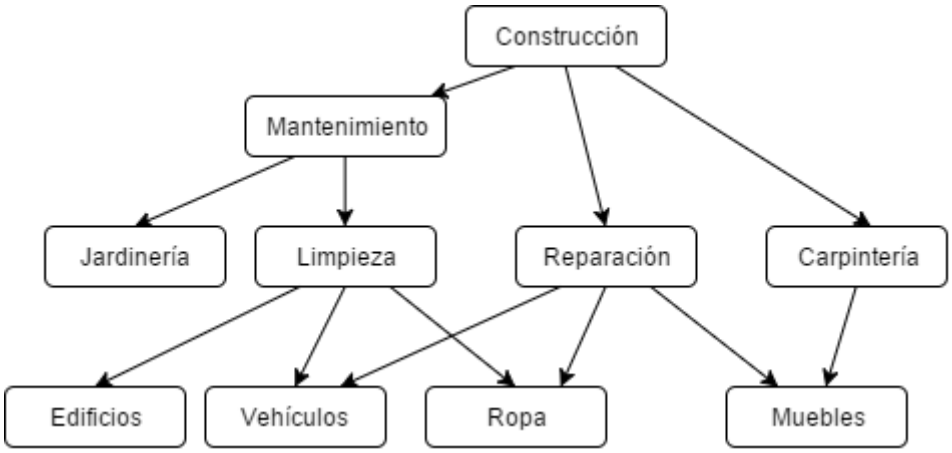

Figura 5.3: Ejemplo de ontología

Para mantener una clasificación de los servicios ofertados en el sistema cada nodo de la red mantendrá una versión propia de la ontología y podrá añadir y quitar nuevas categorías cuando lo desee. En consecuencia, es necesario el intercambio entre usuarios de estas copias de ontologías con el fin de combinarlas y así propagar los cambios de tal manera que el sistema tienda a un estado estable. Por ello, son tres los aspectos que deben ser definidos: parámetros de la ontología, función *merge* (combinación) e intercambio de ontologías.

No obstante, si inicialmente no se parte de una ontología ya detallada, el sistema de clasificación tendrá problemas para llegar a un estado estable, incluso localmente. Por tanto, la posibilidad de introducir cambios en la ontología, aunque sea algo necesario para la aplicación, debería verse como una funcionalidad que se utilizara relativamente poco. Por otro lado, significa que esta ontología inicial debe ser especificada y que debe decidirse si podrá mutar o no, y si lo haría en su totalidad o solo en los niveles más bajos, es decir, sea:

- A_{n0} el conjunto de nodos de la ontología del par n al inicio de la aplicación (en el momento 0)
- A_{ni} el conjunto de nodos de la ontología del par n al en el momento i

existe:

- un conjunto de nodos iniciales $Init \subseteq A_{n0} \; \forall \; n$
- un conjunto de nodos permanentes (no pueden ser borrados por un usuario ni caducan) $Perm \subseteq A_{ni} \; \forall \; n, i$.

Entonces, hay dos posibles configuraciones: $Perm = Init$ o $Perm \subset Init$. No hay otras configuraciones posibles ya que permitir que nodos no-iniciales sean permanentes no tendría mucho sentido. Obsérvese que la propiedad de permanencia no significa que estos nodos no son editables ya que, como se verá más adelante, podría permitirse la ampliación del conjunto de sinónimos que etiquetan un nodo permanente.

Por otro lado, con la idea de ontología dinámica y distribuida que se tiene surge un problema de "seguridad" o "legalidad". ¿Cómo tratar los casos en los que aparecen

categorías relacionadas con temas ilegales? No es una tarea sencilla pues no se goza de un servidor central.

5.5.1.1 Parámetros de la ontología

La ontología estará representada por un DAG en el que los nodos son categorías. Cada uno de estos nodos deberá almacenar la siguiente información:

- **Sinónimos o conjunto de nombres de la categoría a la que se refieren**: se tiene un conjunto de nombres y no solo uno debido a que múltiples categorías en ontologías diferentes se refieren al mismo concepto con diferentes nombres. De este modo no es necesario tener que decidir qué nombre único tendrá cada nodo. También facilita la operación *merge.* Parece razonable que estos sinónimos se vayan añadiendo a medida que se realiza el *merge*, pero es una decisión más de diseño ya que también se podrían incluir todos los sinónimos existentes de una categoría a la vez al crearse un nodo nuevo en la ontología (sinónimos procedentes de alguna base de datos léxica, como por ejemplo WordNet).

- **TTL (Time To Live, tiempo de vida)**: El TTL y el *leasing* son soluciones típicas en los sistemas distribuidos por la falta de conocimiento global. Aunque también se ha optado por asignar un tiempo de vida para evitar un crecimiento incontrolable de la ontología causado por categorías no usadas, el motivo principal es mantener una ontología con las categorías populares del sistema, de tal manera que aquellas que se usan más sobreviven y las que no morirán. Por ejemplo: el TTL de una categoría de la ontología de un par de la red P2P se reseteará cuando el par reciba una petición que la utilice, así como cuando el par oferte un servicio que la usa. Sin embargo, si en el momento de expiración de una categoría existe un servicio clasificado en ella, ésta no podrá expirar hasta que el servicio que depende de ella lo haga también. Si el usuario renueva la oferta del servicio antes de que éste expire entonces también se reseteará el TTL de la categoría. Más adelante se hablará sobre el TTL de los servicios ofertados.

  Por otra parte, hay que considerar de cara al diseño la posibilidad de incluir un **TTL también en las aristas** de la ontología, es decir, las relaciones entre las categorías. Así las clasificaciones de categorías que se usan poco no estarán presentes en la ontología.

5.5.1.2 Estado inicial de la ontología

Para no tener que esperar demasiado a que la ontología llegue a una estructura estable debido a la creación de nuevos nodos por parte de los usuarios, se partirá inicialmente de una versión con múltiples categorías ya definidas. Aquí presentamos nuestra propuesta para esta ontología. Para ello nos hemos apoyado en la clasificación de servicios definida por WIPO teniendo en cuenta qué tipo de servicios se podrían ofrecer en un banco de tiempo, además como la estructura que utiliza es casi plana (2 niveles de jerarquía) se han tenido que añadir nuevos niveles en base a nuestro criterio. En la clasificación que se propone a continuación aparecen categorías repetidas en varios niveles de la estructura, lo que quiere decir que existen una sola vez en la ontología y que son apuntadas por varios nodos padre. Se ha realizado una representación de las clases de la ontología con la herramienta Protégé [81] como se ilustra en la Figura 5.4.

- Construcción, reparación y mantenimiento:
  - Construcción:
    - Carpintería
    - Herrajería
    - Albañilería
    - Cristalería
    - Fontanería
    - Electricidad
    - Calefacción y enfriamiento
  - Mantenimiento:
    - Limpieza:
      - Vehículos
      - Ropa
      - Edificios:
        - Exterior
        - Interior
    - Jardinería:
      - Árboles
      - Otros
    - Horticultura:
      - Plantas comestibles
      - Plantas decorativas:
        - Exterior
        - Interior
    - Edificios:
      - Diseño de interiores
      - Estudios energéticos
      - Estudios estructurales
  - Reparación:
    - Hardware
    - Ropa:
      - Calzado
      - Otra ropa
    - Vehículos:
      - Motorizados:
        - Aviones
        - Patinetes eléctricos
        - Quads
        - Coches y furgonetas
        - Camiones y autobuses
        - Otro
      - No motorizados:
        - Bicicletas
        - Monopatines
    - Muebles
- Transporte y almacenamiento:
  - Transporte:
    - Personas

        - Mercancía
    - Entrega:
        - Cartas
        - Paquetes
    - Almacenamiento:
        - Guardamuebles
- Educación, cultura y deporte:
    - Clases:
        - Infantil
        - Primaria
        - ESO
        - Bachillerato
        - Universidad
        - Formación Profesional
        - Deporte y salud
        - Hobbies
        - Otro
    - Ocio:
        - Organización de eventos
        - Espectáculo
        - Turismo:
            - Rutas
- Tratamiento de productos:
    - Imagen:
        - Fotografía:
            - Sesión
            - Edición
        - Vídeo:
            - Sesión
            - Edición
    - Textil
- Cuidado:
    - Personas:
        - Compañía y cuidados generales:
            - Ancianos
            - Niños
            - Otros
        - Salud:
            - Tratamientos médicos y dentales
            - Tratamientos terapéuticos
            - Tratamientos estéticos
    - Animales:
        - Tratamientos veterinarios
        - Compañía y cuidados generales
- Tecnología:
    - Software:
        - Instalación
        - Actualización
        - Asesoramiento
    - Hardware:

        - Instalación
        - Asesoramiento
- Uso temporal de bienes:
    - Locales:
        - Uso residencial
        - Uso comercial
    - Vehículos:
        - Motorizados
        - No motorizados
    - Herramientas:
        - Eléctricas
        - De gasolina
        - Otras
    - Instrumentos musicales
    - Aparatos tecnológicos
    - Otros

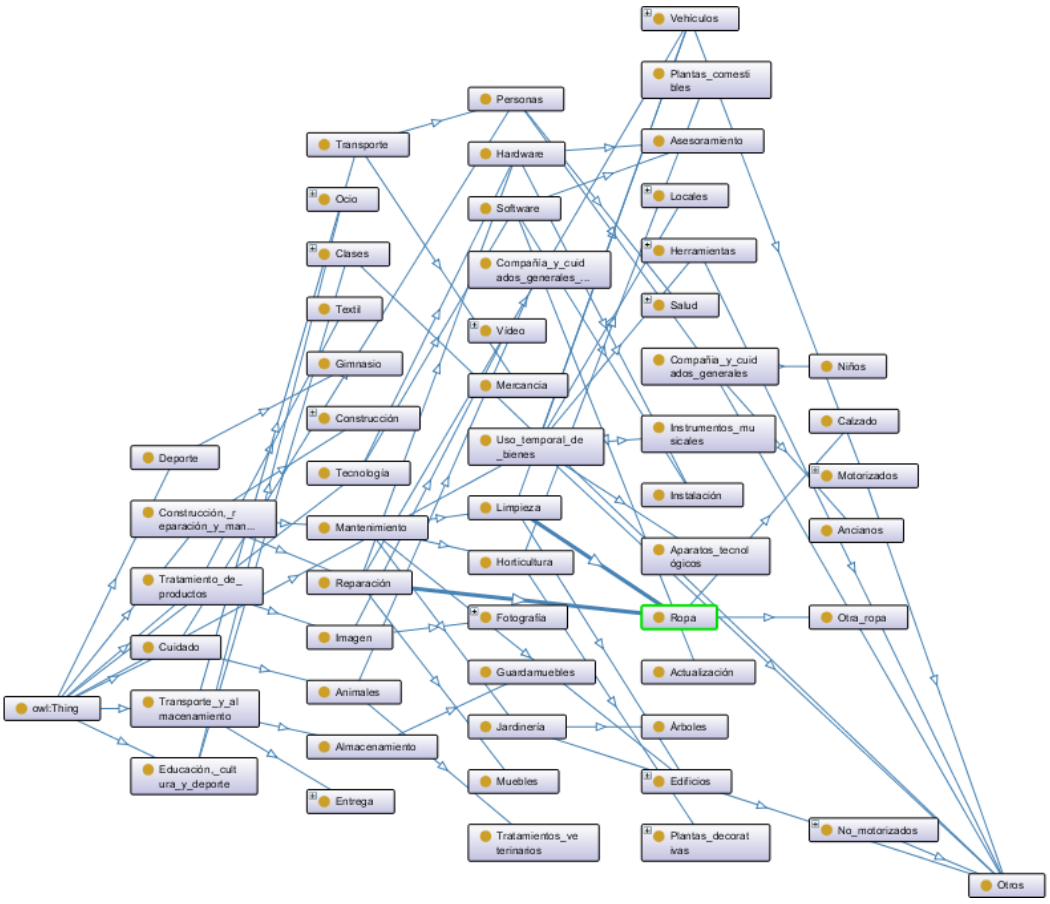

Figura 5.4: Representación con Protégé de la ontología inicial parcialmente expandida

5.5.1.3 Función *merge*

Para que el sistema de oferta y demanda de servicios funcione correctamente, el papel que juega la función de combinación de ontologías es vital. De ella depende principalmente que cada nodo mantenga una versión parecida a la de otro para que exista consistencia en la solicitud de servicios. Sin embargo, ya que los elementos que forman la ontología no siguen un orden (cardinal, cronológico, alfabético, etc.), esta tarea se vuelve compleja y no trivial. El algoritmo de combinación será lo más automático posible, no obstante habrá algunos casos en los que será necesaria la intervención del usuario para resolver conflictos o situaciones de ambigüedad, pero ésta debe ser la última de las alternativas a las que acudir.

En un principio la estructura de datos escogida fue un árbol por sencillez de procesamiento, sin embargo, finalmente se decidió usar un DAG por las ventajas que ofrece. A continuación se muestran ejemplos sobre esta decisión. Los ejemplos tratan árboles y posteriormente se mostrará cómo se solucionarían los problemas mediante un DAG.

En estos casos de ejemplo la función toma como entrada dos árboles (t_1 y t_2) que representan dos ontologías, y devuelve una nueva ontología también en forma de árbol (t_r), resultado de combinar las dos primeras.

Como la solución no es trivial, se van a analizar algunos casos sobre cuál debería ser el resultado de la función. Los árboles se han dibujado utilizando la herramienta web para construir grafos draw.io [86].

Caso 1:

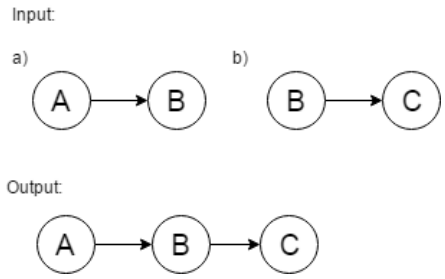

Figura 5.5: Merge caso 1

Caso 2:

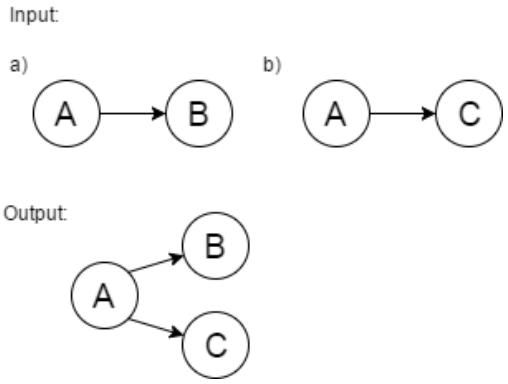

Figura 5.6: Merge caso 2

Caso 3: En este caso puede haber dos soluciones ya que se desconoce a priori la relación entre A y B.

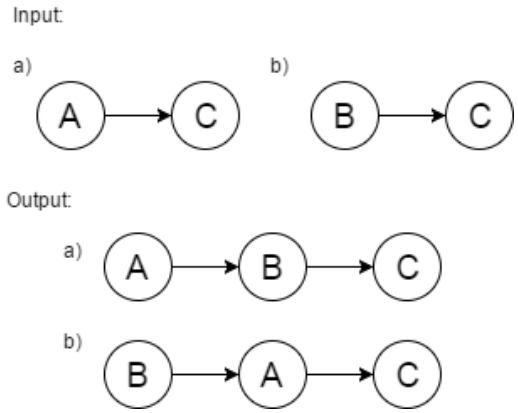

Figura 5.7: Merge caso 3

Caso 4: En este caso puede haber dos soluciones ya que se desconoce a priori si C es más general que B o si están ambas al mismo nivel.

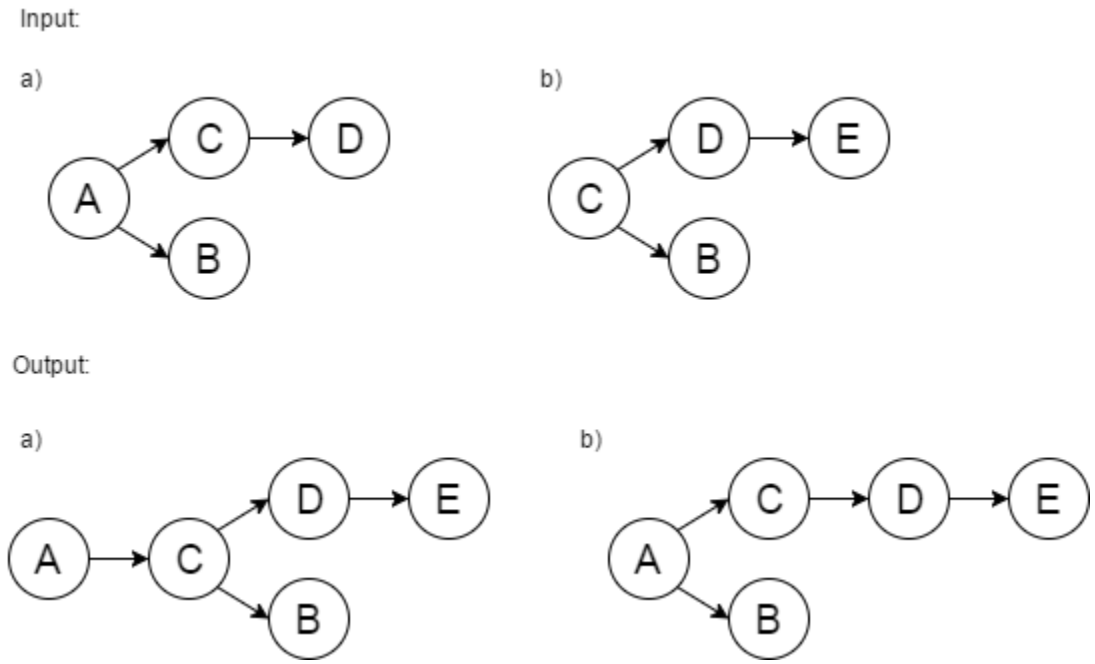

Figura 5.8: Merge caso 4

Los casos uno y dos son simplemente para mostrar algunos de los casos más obvios y directos. El tercer caso es un ejemplo para mostrar los problemas sobre obligar a tomar decisiones al usar un árbol. Como el algoritmo no conoce cuál es la verdadera relación de parentesco entre las categorías A y B surgen dos posibles soluciones, pero como solo se debe producir un resultado surge la necesidad de idear un mecanismo para resolver esta ambigüedad. Sin embargo, en esta situación no es necesario complicar la solución con mecanismos extra si en lugar de utilizar árboles se usan DAGs. Con un DAG se puede resolver el conflicto con una única solución sin tener que tomar ninguna decisión como se muestra en la Figura 5.9.

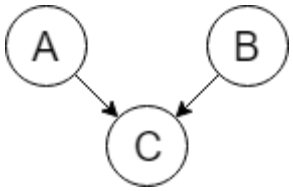

Figura 5.9: Solución Merge caso 3

En el cuarto caso también se produce una única solución al usar un DAG como se ilustra en la Figura 5.10. Pero esta vez la ontología resultante quizás se podría reestructurar y simplificarse (Figura 5.10) ya que el nodo C es más general que B pero más particular que A, y conviene mantener la estructura lo más lineal y arborescente posible aunque sean DAGs las estructuras que se traten

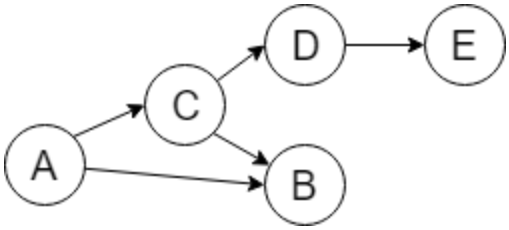

Figura 5.10: Solución Merge caso 4

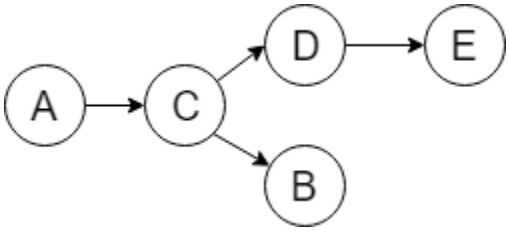

Figura 5.11: Solución alternativa Merge caso 4

El siguiente problema a abordar es cómo construir el DAG resultante a partir de los dos de entrada. El mecanismo ideado consiste inicialmente en formar un conjunto de pares de vértices formado por las aristas de los grafos de entrada, y posteriormente reconstruir un nuevo grafo a partir de un conjunto total o parcial de ellas. Usando este mecanismo se tiene el riesgo de encontrar bucles en el grafo final. Para ello se debe incluir un algoritmo de detección de ciclos. El siguiente caso ejemplifica un proceso de construcción de la solución usando este método:

1. Las ontologías de entrada son las mismas usadas en el caso cuatro (Figura 5.8). Por lo que el primer paso es formar el conjunto de pares de vértices o aristas (Figura 5.12).
2. El siguiente paso es usar los pares obtenidos y unificarlos en un solo grafo. Para ello se recorrerá el conjunto de aristas y se irán conectando evitando repeticiones de aristas ya incluidas y ciclos, para dar lugar a un DAG (Figura 5.13).
3. Por último sería necesario procesar el grafo para eliminar las aristas redundantes.

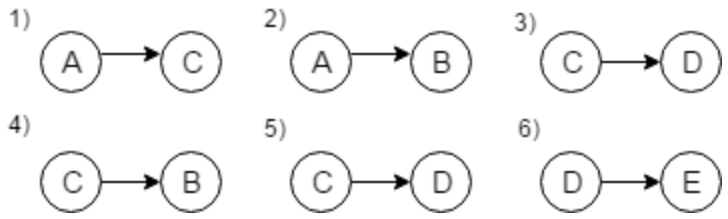

Figura 5.12: Merge ejemplo completo 1 paso 1

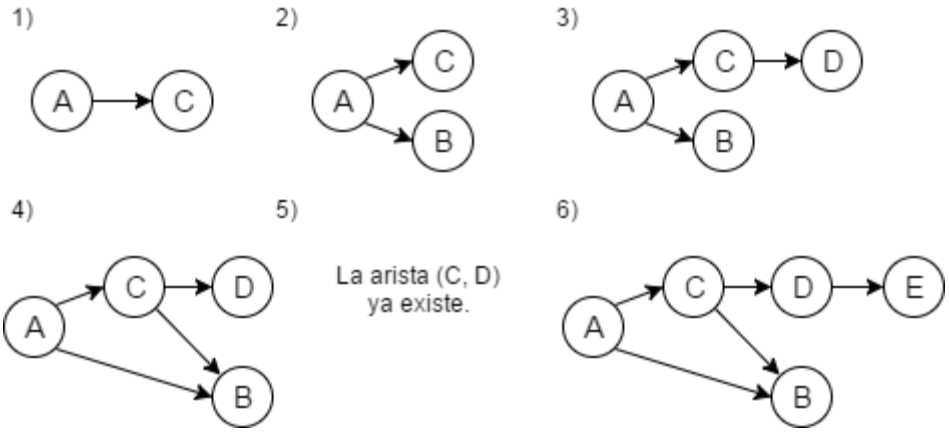

Figura 5.13: Merge ejemplo completo 1 paso 2

También sería conveniente describir una situación en la que se produzcan bucles y ver cómo actuar ante tal hecho:

1. Al igual que antes, se parte de dos ontologías (Figura 5.14).
2. Se forma el conjunto de pares (Figura 5.15).
3. Tras ello ya se procede a construir el grafo final (Figura 5.16).
4. Al insertarse la última arista un ciclo es detectado, formado por los vértices A, C y D.
5. Tras ello surge una de las situaciones en las que la función puede necesitar la **intervención del usuario** para resolver la ambigüedad. Se debe escoger qué arista eliminar entre las tres que forman el bucle. Una solución puede ser eliminar la arista que conecta D con A.
6. Hasta que no se solucione el conflicto, la ontología no se propagará a otros vecinos.

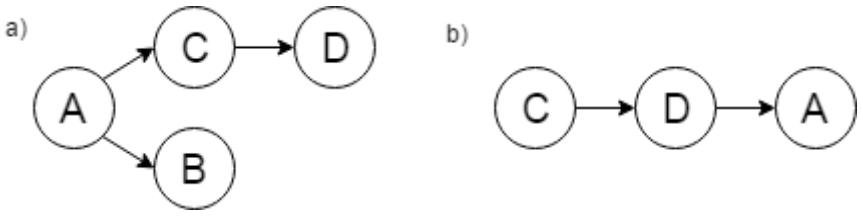

Figura 5.14: Merge ejemplo completo 2 entrada

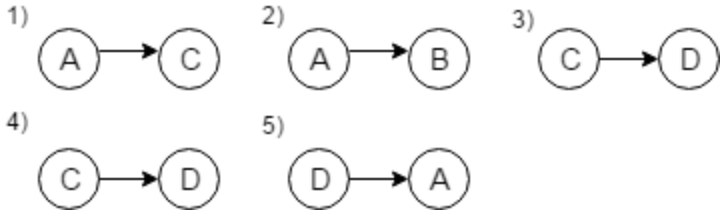

Figura 5.15: Merge ejemplo completo 2 paso 1

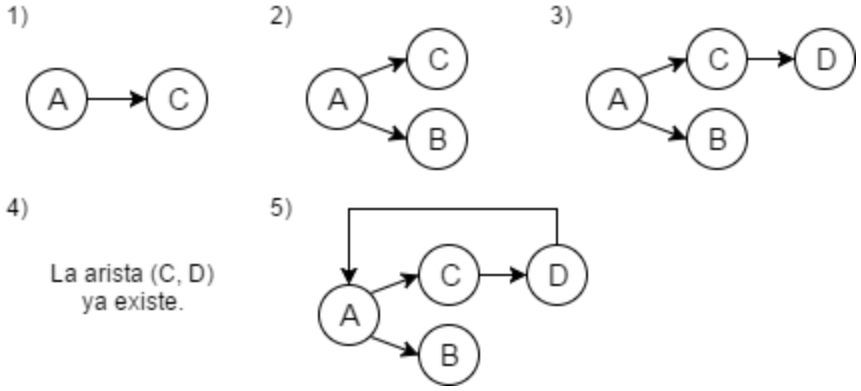

Figura 5.16: Merge ejemplo completo 2 paso 2

Otro punto crucial es determinar cómo comparar los nodos de la ontología debido a que son los propios usuarios los que crean las categorías del DAG y en consecuencia puede haber nodos que se refieran al mismo concepto y sin embargo estar escritos de diferente manera. Puede haber varios casos por los que esto ocurra: Uso de **sinónimos** y frases con el mismo valor semántico o **errores ortográficos**. Sin un mecanismo para detectar estos casos, la ontología estaría compuesta por categorías repetidas ya que a ojos de una comparación por igualdad de nombre son diferentes. Para lidiar con este problema se ha propuesto consultar en alguna base de datos léxica la existencia de sinonimia entre dos categorías del DAG. Un ejemplo de base de datos léxica es WordNet [53]. De este modo, cuando la función *merge* encuentre en una ontología el término "Cuidado de mayores" y en otra el término "Cuidado de ancianos", deberá ser capaz de reconocerlos como iguales.

Antes de indagar en la especificación del algoritmo de *merge* se dedicará un apartado a definiciones formales acerca de la teoría de grafos.

5.5.1.4 Definiciones formales

5.5.1.4.1 Definiciones básicas de la teoría de grafos

Un *grafo* es una 2-tupla (V, E) donde:

- V es un conjunto de vértices (también llamados nodos).
- $E \subseteq (V \times V)$ es un conjunto de aristas.

Un *camino en un grafo* es una secuencia de vértices tal que existe una arista entre cada vértice y el siguiente. Un *ciclo* en un grafo es un camino tal que existe una arista entre el

último vértice y el primero. Un grafo es *conexo* si existe un camino entre dos vértices cualesquiera. Un *árbol* es un grafo conexo y acíclico.

Un *grafo con nodos etiquetados* es una 3-tupla (V, E, α) donde:

- (V, E) es un grafo.
- $\alpha: V \rightarrow L$ es una función que asigna una etiqueta del conjunto de etiquetas L a cada uno de los vértices. Si α es inyectiva, decimos que el grafo es un *grafo con nodos distinguidos*.

Un *grafo con nodos multi-etiquetados* es una 3-tupla (V, E, α) donde:

- (V, E) es un grafo.
- $\alpha: V \rightarrow \mathcal{P}(L)$ es una función que asigna un subconjunto del conjunto de etiquetas L a cada uno de los vértices. Si α tiene la propiedad de que $\forall\ v_1, v_2 \in V, \alpha(v_1) \cap \alpha(v_2) = \emptyset$ entonces decimos que el grafo es un *grafo con nodos multi-distinguidos*.

Un *grafo dirigido* (con o sin nodos etiquetados o multi-etiquetados) es un grafo en el que las aristas son pares ordenados; el primer componente del par ordenado se denomina la *cola* de la arista y el segundo, la *cabeza*. Decimos que el vértice cola es un *predecesor* del vértice cabeza y el vértice cabeza es un *sucesor* del vértice cola. El *grafo subyacente* de un grafo dirigido es el grafo que se obtiene por medio de sustituir las aristas dirigidas por aristas no dirigidas, es decir, por perder la información de orden entre los vértices de cada arista. Un *camino dirigido* en un grafo dirigido (con o sin nodos etiquetados o multi-etiquetados) es una secuencia de aristas tal que la cabeza de cada arista, salvo la de la última, es la cola de la arista siguiente. Un *ciclo en un grafo dirigido* (con o sin nodos etiquetados o multi-etiquetados) es un camino en el que la cabeza de la última arista es la cola de la primera. Un grafo dirigido es *débilmente conexo* si el grafo subyacente es conexo. Un grafo dirigido es *fuertemente conexo* si hay un camino dirigido entre dos vértices cualesquiera.

Por otro lado son de especial importancia los *grafos dirigidos acíclicos* (con o sin nodos etiquetados o multi-etiquetados), conocidos como DAG ("directed acyclic graph"). Un DAG es *arborescente* si cada vértice tiene un solo predecesor. Un vértice de un DAG es una *raíz* si no tiene ningún predecesor. Un *DAG de raíz única* es un DAG que contiene una sola raíz. Obsérvese que si un DAG tiene una sola raíz, cada nodo es alcanzable desde la raíz (ver la noción de *ordenación topológica*). Un poliárbol es un DAG cuyo grafo subyacente es un árbol. Un *árbol dirigido* es un DAG de raíz única y arborescente. Si la etiqueta de la raíz de dos DAG con nodos distinguidos/multi-distinguidos de raíz única es la misma (o, más general, equivalente), decimos que los DAG tienen *raíz (única) común*.

Para representar nuestras ontologías, nos interesan particularmente los DAG conexos con nodos distinguidos/multi-distinguidos y con raíz única o, quizás, algún subtipo de este tipo de DAG, por ejemplo los árboles dirigidos con nodos distinguidos/multi-distinguidos. Además, vamos a trabajar con familias de DAGs que tienen la característica de tener raíz (única) común.

Una noción de *merge* de dos grafos es de gran importancia para nuestra aplicación.

5.5.1.4.2 *Merge* de grafos

Un *merge* de dos grafos A y B se puede definir como:

1. Encontrar el subgrafo común máximo M de A y B.

2. Añadir los vértices y aristas de $A \setminus M$ y de $B \setminus M$ a M.

   En el caso en el que los grafos A y B tengan propiedades particulares, puede ser de interés que el resultado del *merge* tenga las mismas propiedades, por lo que el *merge* puede implicar un tercer paso:

3. Modificar el grafo resultante, por ejemplo desechar un conjunto mínimo de aristas y vértices, para que tenga las propiedades requeridas. Nótese que, en el caso general, no hay una única solución a este problema.

   En el caso de interés aquí, los grafos son familias de DAG conexos, de raíz única común y con nodos multi-distinguidos, o algún subtipo de este tipo, y el resultado del *merge* debe ser del mismo tipo.

Sobre el paso 1:

El primer paso se ha estudiado desde hace muchos años (más específicamente el "maximum common induced subgraph problem" y el "maximum common edge subgraph problem") y está bien demostrado que el problema general es NP-completo (aunque polinomial para árboles con ciertas restricciones). Sin embargo, casi todos estos estudios conciernen a grafos no etiquetados, en cuyo caso el subgrafo común máximo se tiene que definir en términos de subgrafos isomorfos (de allí la conexión con el "subgraph isomorphism problem"). En el caso de grafos con nodos etiquetados, y aún más en el de grafos con nodos distinguidos/multi-distinguidos, el problema se simplifica mucho.

Sobre los pasos 1 y 2:

Si los grafos tienen nodos etiquetados, podemos realizar los pasos 1 y 2 con un "*merge* basado en etiquetas" como sigue (también podríamos simplificar y suponer que el conjunto de etiquetas es el mismo para todos los grafos).

El resultado del *merge* de dos grafos etiquetados $G_1 = (V_1, E_1, \alpha,: V_1 \rightarrow L_1)$ y $G_2 = (V_2, E_2, \alpha_2: V_2 \rightarrow L_2)$ es el grafo $G_r = (V_r, E_r, \alpha_r : V_r \rightarrow L_1 \cup L_2)$ donde:

- $V_r = (V_1 \cup V_2) / EQ$, es el conjunto cociente, donde EQ es la relación de equivalencia $EQ(u, v) \leftrightarrow EQ\text{-}L(\alpha_1(u), \alpha_2(v))$, siendo $EQ\text{-}L$ una relación de equivalencia sobre $L_1 \cup L_2$.

  Identificamos a cualquier clase de equivalencia de vértices que contiene un solo elemento con el elemento contenido.

→ Es decir, los vértices u de G_1 y v de G_2 que tienen etiquetas equivalentes se representan con un solo vértice en G_r, y cada vértice de G_1 o de G_2 por el que no existe un vértice en el otro grafo con una etiqueta equivalente se representa con un vértice correspondiente en G_r.

- $E_r = \{(a_{r1}, a_{r2}) \mid a_{r1}, a_{r2} \in V_r : \exists\ a_1 \in a_{r1}, a_2 \in a_{r2} : ((a_1, a_2) \in E_1 \vee (a_1, a_2) \in E_2)\}$.

  En el caso de grafos dirigidos los pares de vértices, tanto de G_1 y G_2 como de G_r están ordenados.

  → Es decir, el conjunto de aristas de G_r es la unión de los conjuntos de aristas de G_1 y G_2, tomando en cuenta la relación de equivalencia entre vértices.

- α_r se define como sigue:

  $\alpha_r(v) = \alpha_1(v_1)$, si $v = \{v_1\}$ para algún $v_1 \in V_1$

  $\alpha_r(v) = \alpha_2(v_2)$, si $v = \{v_2\}$ para algún $v_2 \in V_2$

  $\alpha_r(v) = \alpha_1(v_1) \vee \alpha_2(v_2)$, si $v = \{v_1, v_2\}$, para algún $v_1 \in V_1$ y $v_2 \in V_2$

  → Es decir, un vértice de G_r se etiqueta con cualquiera de las dos etiquetas equivalentes, si se trata de un vértice derivado de vértices equivalentes en G_1 y G_2, y con la etiqueta del vértice con el que se identifica, en caso contrario.

Sin embargo, en el caso de encontrar un nodo en G_1 equivalente pero no igual a un nodo en G_2, es bastante insatisfactorio tener que elegir cuál de las etiquetas equivalentes se va a usar para etiquetar el nodo en G_r. Mejor sería etiquetar el nodo en G_r con las dos etiquetas equivalentes. Para ello, tenemos que definir un *merge* entre grafos con nodos multi-etiquetados. El resultado del *merge* de dos grafos multi-etiquetados $G_1 = (V_1, E_1, \alpha_1: V_1 \rightarrow \mathcal{P}(L_1))$ y $G_2 = (V_2, E_2, \alpha_2: V_2 \rightarrow \mathcal{P}(L_2))$ es el grafo $G_r = (V_r, E_r, \alpha_r: V_r \rightarrow \mathcal{P}(L_1 \cup L_2))$ donde:

- $V_r = (V_1 \cup V_2) / EQ$, es el conjunto cociente, donde EQ es la relación de equivalencia $EQ(u, v) \leftrightarrow EQ\text{-}\mathcal{P}(L)(\alpha_1(u), \alpha_2(v))$, siendo $EQ\text{-}\mathcal{P}(L)$ la relación de equivalencia sobre $\mathcal{P}(L_1 \cup L_2)$ inducido por $EQ\text{-}L$, una relación de equivalencia sobre $L_1 \cup L_2$,como sigue:

  $(L_1, L_2) \in EQ\text{-}\mathrm{P}(L)$ si $\exists\ l_1 \in L_1 \subseteq L \wedge l_2 \in L_2 \subseteq L$: $(l_1, l_2) \in EQ\text{-}L$.

  Identificamos a cualquier clase de equivalencia de vértices que contiene un solo elemento con el elemento contenido.

  → Es decir, los vértices u de G_1 y v de G_2 cuyos conjuntos de etiquetas solapan se representan con un solo vértice en G_r, y cada vértice de G_1 o de G_2 por el que no existe un vértice en el otro grafo con un conjunto de etiquetas que solapa se representa con un vértice correspondiente en G_r. Decimos que hay solape entre dos conjuntos de etiquetas si tienen elementos equivalentes.

- E_r. Las aristas se definen igual que en el caso de grafos etiquetados.

- α_r: $V_r \rightarrow P(L_1 \cup L_2)$ se define como sigue:

  $\alpha_r(v) = \alpha_1(v_1)$, si $v = \{v_1\}$ para algún $v_1 \in V_1$.

  $\alpha_r(v) = \alpha_2(v_2)$, si $v = \{v_2\}$ para algún $v_2 \in V_2$.

  $\alpha_r(v) = \alpha_1(v_1) \cup \alpha_2(v_2)$, si $v = \{v_1, v_2\}$ para algún $v_1 \in V_1$ y $v_2 \in V_2$

Obsérvese que estas definiciones dependen de la relación de equivalencia entre etiquetas (la relación *EQ-L*), que no se define aquí, y que con estas definiciones:

- Si G_1 y G_2 son grafos con nodos distinguidos (respectivamente multi-distinguidos), G_r también lo es.
- Si G_1 y G_2 son grafos dirigidos conexos con raíz única, G_r también lo es, y si, además, G_1 y G_2 tienen raíz común, G_r también la tiene.

Sobre el paso 3:

Para este paso existen múltiples soluciones dependiendo del tipo de grafo escogido:

1. DAGs conexos de raíz única común y con nodos distinguidos/multi-distinguidos:

   Con las definiciones dadas, la dificultad en asegurar que el resultado del *merge* de dos DAG de este tipo también sea de este tipo reside en asegurar que el resultado sea un DAG, es decir, que sea acíclico. Se puede ver fácilmente que, para conseguirlo, basta con desechar un conjunto mínimo de aristas del grafo resultado del merge basado en etiquetas. Este problema se conoce como el *problema del conjunto de vértices de realimentación* o el *problema del conjunto de aristas realimentación* y se ha demostrado que, en el caso general, es un problema NP-completo. Sin embargo, en nuestro caso, los grafos son pequeños y tratables.

2. Árboles dirigidos con raíz común y con nodos distinguidos/multi-distinguidos:

   Para asegurar que el resultado del *merge* de dos árboles dirigidos sea otro árbol dirigido, además de garantizar que el resultado sea acíclico, se tiene que garantizar que sea arborescente. Otra vez se puede ver fácilmente que, para conseguirlo, basta con desechar un conjunto mínimo de aristas del grafo resultado del merge basado en etiquetas. Este problema se reduce al cálculo de la *reducción transitiva*, el grafo con el mismo número de aristas y la misma relación de alcanzabilidad (que en el caso de un DAG finito es único y siempre es un subgrafo del grafo original). Se construye descartando aristas entre dos vértices conectados también por otro camino más largo. Hay algoritmos de orden $O(mn)$ para la reducción transitiva donde m es el número de vértices y n el número de aristas.

Algoritmos requeridos

Ahora que se tiene una aproximación más concreta de lo que se busca, se pueden estudiar los algoritmos que hacen falta para esta tarea. En función de los problemas que se han detectado se tienen:

- **Clean-up**: El objetivo de este algoritmo es borrar aquellas aristas consideradas redundantes como ocurría en el caso de ejemplo cuatro del *merge* resuelto con un DAG (Figura 5.10 y Figura 5.11). El problema que surge ahora es asegurar que no complica la tarea de quitar un nodo no popular de la ontología (su TTL expira), es decir, se tiene que mirar cuidadosamente su interacción con el efecto del TTL en los nodos y con ello definir en qué situaciones se resetea el TTL y en cuáles disminuye y expira.

  Por otro lado, este algoritmo guarda relación con el algoritmo que matchea el camino enviado como parámetro de una petición de búsqueda, con el DAG de la ontología del receptor de la petición. Si la ontología tiene forma de DAG, en vez de un camino, el parámetro de la búsqueda podría ser un subgrafo, pero esto es una decisión de diseño. Con esta alternativa surgen complicaciones en la eficiencia pues hay que realizar más comprobaciones pero también existen ventajas, ahora aparecerán más resultados en la búsqueda pues se consultan más tipos de servicios. Este algoritmo es similar al de *merge* pero con la complicación de tener que resetear los TTL de los nodos usados en la petición.

- ***Subsumption***: el término *subsumption* hace referencia a la relación de generalización entre dos objetos, donde un primer objeto es más general que un segundo (hiperónimo), y el segundo es más particular que el primero (hipónimo). En el caso que concierne a la función *merge* podemos aplicar la relación de generalización que pueda haber entre dos categorías de la ontología para obtener una estructura más coherente, en el sentido de que un nodo más general debería ser subnodo de un nodo más específico. Para ello WordNet ofrece una base de datos léxica con información de sobre hiponimia e hiperonimia. Así pues distinguimos dos casos donde se puede aplicar esto. Para simplificar nos referiremos a árboles en ejemplos acompañados de pseudocódigo:

  - Para dos nodos hijo de un nodo *g* de Gr, *a* que viene de G_1 y *b* que viene de G_2:

```
if is_hyponym(a, b)
      if is_hyponym(a, b1)      donde b1 viene de G2 y es un
                                nodo hijo de b
            if is_hyponym(a, b2)      donde b2 viene de G2 y es
                                      un nodo hijo de b1
                  ...
                  Se cuelga el árbol con raíz a como hijo de bn
            else
                  Se cuelga árbol con raíz a como hijo de b1
      else
            Se cuelga árbol con raíz a como hijo de b

else if is_hyponym(b, a)
      if is_hyponym(b, a1)      donde a1 viene de G2 y es un
                                nodo hijo de a
            ...
            Se cuelga el árbol con raíz b como hijo de an

      else
            Se cuelga árbol con raíz b como hijo de a
else
      Se deja a y b como dos hijos de g
```

- Además para el caso de un DAG, para dos nodos padre de un nodo *g* de un G_r, *a* que viene de G_1 y *b* que viene de G_2:

```
if is_hyponym(a, b)
      Se cuelga el subgrafo con raíz a como hijo de b.
else if is_hyponym(b, a)
      Se cuelga el subgrafo con raíz b como hijo de a.
else
      Se deja a y b como dos padres de g.
```

- **Expiración del TTL**: En este caso, son muchas las consideraciones a tener en cuenta pues la solución no es única. Surgen de este modo varias cuestiones:
  - En relación a qué nodos pueden expirar: ¿Cuáles son los nodos que pueden expirar? ¿El conjunto de nodos que no pueden expirar coincide con el conjunto de nodos inicial o es un subconjunto de ellos?
  - En relación a qué aristas pueden expirar (si alguna), ¿hay relación entre el TTL de una arista y el de los nodos que conecta? ¿deben ser los TTLs de las aristas más cortos que los de los nodos? No se debe permitir la expiración de una arista si dejaría el grafo inconexo.
  - En relación al valor del TTL: ¿Qué medida usar para el TTL? ¿Cuál es un valor razonable para el TTL de un nodo de la ontología? ¿El valor es el mismo para todos los nodos (que pueden expirar)? ¿O usamos un valor más alto para los nodos (que pueden expirar) que están más cerca de la raíz? Por ejemplo, el TTL de un nodo podría ser una función del TTL de sus nodos hijo, un candidato obvio siendo la función *maximum*. ¿El valor del TTL es proporcional al tamaño del sistema (p.ej. número de usuarios)? ¿Cómo se pueden tener localmente medidas sobre el tamaño global del sistema?

Muchas de estas preguntas sólo pueden responderse mediante simulación ya que a priori no se puede afirmar por ejemplo, el valor inicial del TTL para un nodo

sin haberlo probado antes y haber comprobado que encaja con el funcionamiento global de la aplicación.

5.5.1.5 Intercambio de Ontologías

El papel que juega la combinación de versiones de ontologías es crucial pero para ello ha tenido que haber antes un envío de éstas entre dos nodos de la red. Aquí se definen las distintas partes que intervienen en este envío:

- Contenido del mensaje: aparte de la ontología, qué información extra será necesario transmitir.
- Destino del mensaje: a quién se enviará el mensaje (solo a vecinos, todos los nodos posibles en un tiempo de vida del mensaje, etc.).
- Periodicidad: cada cuánto se realiza el envío de ontologías.
- Cómo se enviará: como mensaje explícito para ello o aprovechando el envío de otro tipo de mensajes (*piggy back*).

5.5.2 Oferta de servicios

En este punto se describe el proceso de cómo los usuarios ofertarán servicios en el banco de tiempo para que otros los puedan buscar. La idea que se busca es mantener los servicios clasificados en categorías del mismo modo que en otras aplicaciones famosas ya mencionadas.

En este punto se tratarán las dos posibilidades relacionadas con la oferta de servicios: publicación de servicios y eliminación de los mismos.

5.5.2.1 Publicación de servicios

Como ya se adelantó en la idea base, cada usuario mantiene una versión local de la ontología de categorías de servicios. Cuando nos referimos al término local no necesariamente tiene que estar ubicada en el propio nodo, sino que puede estar también ubicada en un supernodo si la red P2P lo soporta, pero quien realmente tiene el control sobre ella sí que es el propio usuario. Pues bien, cuando un usuario concreto desee publicar un servicio en el banco de tiempo, lo hará pero de manera local:

1. Se le asignará al servicio una categoría de la versión local de la ontología.
2. La información de este nuevo servicio se almacenará únicamente en el propio nodo del usuario de la red o también en un supernodo.
3. Si no existe una categoría que según el usuario no se adapta al servicio que quiere publicar podrá entonces añadir una nueva categoría como nodo de la ontología. Condiciones:
    a. Cuando se introduzca un nombre para la nueva categoría se buscará esta palabra o frase en alguna base de datos léxica para comprobar su existencia y ortografía. La finalidad de este paso es mantener una ontología bien formada y reducir la complejidad de la comparación de categorías en la función *merge*.

b. La nueva categoría podrá ser añadida a la ontología de manera que cuelgue de uno o más nodos de ésta, pero se comprobará que no se forma ningún ciclo ya que de ser así no se permitirá su inclusión.
c. No podrá haber repeticiones, aunque si el usuario lo considera, en lugar de añadir una nueva categoría, puede unir dos ya existentes. También se permite la eliminación de categorías y de relaciones entre ellas.

Para entender mejor el proceso, se ejemplificará con dos casos de uso, en uno el usuario usará una categoría existente y en el otro caso creará una nueva categoría en la ontología.

Caso de uso 1:

Supongamos una ontología con el estado siguiente almacenada en un determinado par de la red:

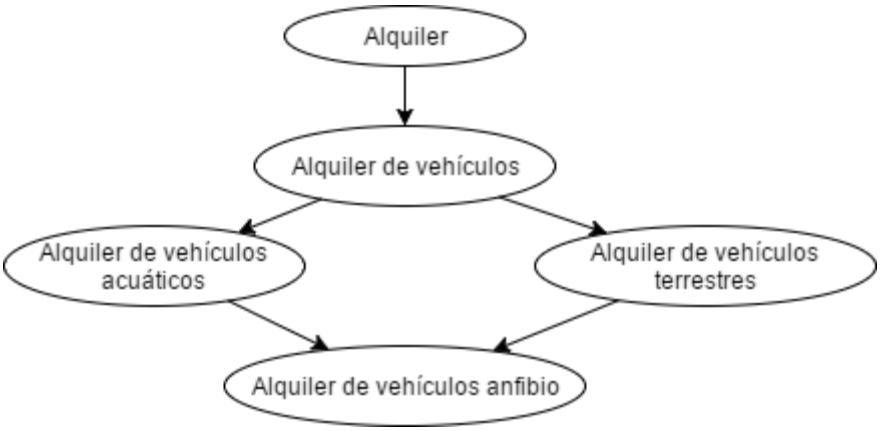

Figura 5.17: Oferta de servicios caso de uso 1 ontología inicial

El usuario de este par quiere ofrecer como servicio el alquiler de su vehículo anfibio. Como las ontologías del sistema están representadas mediante un DAG, puede haber varios caminos hasta una raíz desde un vértice del grafo. Es por eso que el usuario deberá escoger uno de los posibles caminos desde la raíz hasta el nodo "Alquiler de vehículos anfibio" (o bien el subgrafo que contiene todos ellos). En este caso, al tratarse de un servicio relacionado con alquiler de vehículos tanto acuáticos como terrestres, el usuario decide seleccionar todas las rutas hasta la raíz:

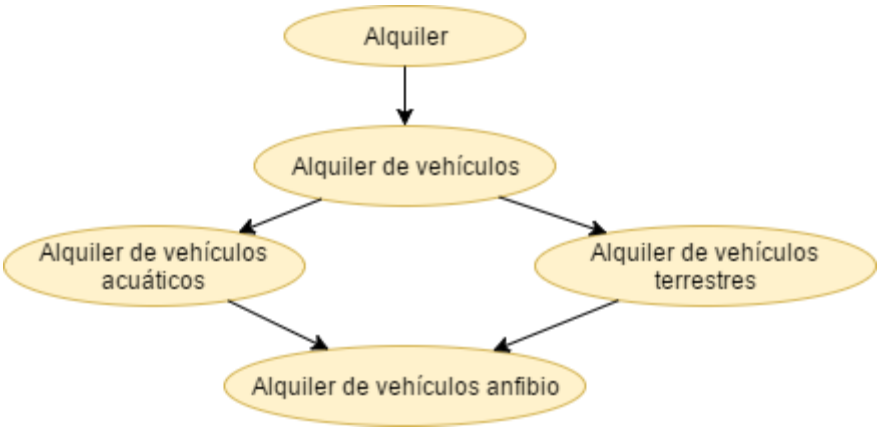

Figura 5.18: Oferta de servicios caso de uso 1 camino elegido

Esta ruta se asocia al resto de datos que debe haber sobre un servicio (ej.: descripción, usuario ofertante, requisitos, etc.) y una vez completada esta información se almacena de manera local para el usuario.

Caso de uso 2:

Ahora supongamos un segundo posible caso en el que el usuario en lugar de querer dar clases particulares de Windows se ofrece para la reparación de bicicletas. Tomando la misma ontología anterior el usuario observa que no existe la categoría “Bicicletas”, por lo que crea un nuevo nodo en el grafo llamado “Bicicletas”:

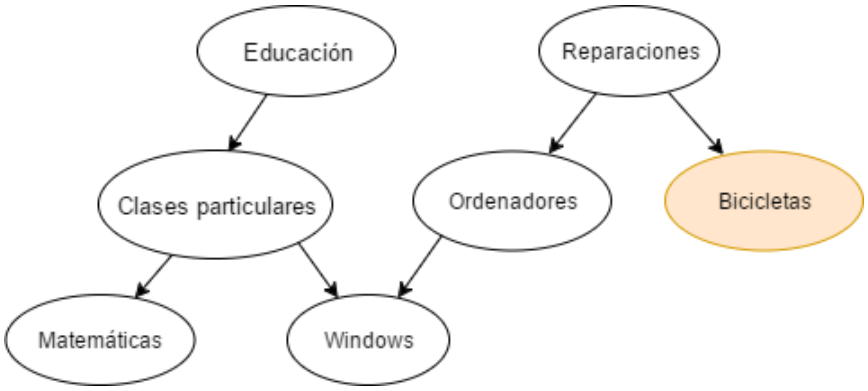

Figura 5.19: Oferta de servicios caso de uso 2 nodo creado

El criterio para ubicar una nueva categoría depende únicamente del usuario, es decir, tiene total libertad, por lo que se podría haber insertado sin tener que ser un nodo hoja. Tras añadirse esta categoría el proceso es idéntico al descrito en el caso de uso anterior.

Otro problema a tratar con esta solución y que no es propio de esta aplicación sino que es una cuestión típica de las redes P2P, es el siguiente: qué hacer cuando un usuario ofertante de un determinado servicio se desconecta de la red temporalmente y posteriormente otro usuario realiza un búsqueda de un servicio de ese mismo tipo. Como es de esperar que un usuario no esté constantemente en línea, es preferible que los servicios de un usuario desconectado sigan apareciendo como resultados en las búsquedas. Es una cuestión que depende mucho del tipo de red que se use, por eso, se volverá a este tema más adelante cuando se hable de la infraestructura de red elegida.

5.5.2.2 Eliminación de servicios

Un usuario debe ser capaz de dejar de ofrecer en cualquier momento cualquier servicio de los que haya publicado previamente. Esta tarea es sencilla pues el control sobre la información de los servicios lo tendrá siempre el usuario y para ello bastaría con eliminarla.

Sin embargo, aparte de poder eliminarse los servicios de manera explícita, éstos tendrán asignado un TTL (Time To Live) el cual una vez expirado el servicio será suprimido. La razón por la cual existe este tiempo de vida es para obligar a los usuarios a mantener actualizado el conjunto de servicios publicados y no acumular ofertas que han caído en el olvido por parte de los ofertantes pero que siguen siendo solicitadas. Si un servicio está clasificado dentro de una categoría cuyo TTL ha expirado pero el del servicio no, la categoría no será eliminada mientras siga vigente el servicio.

5.5.3 Demanda de servicios

La otra parte implicada es la demanda de servicios y al igual que antes, también se hará uso de la ontología y las rutas desde la raíz hasta las categorías clasificadas en ella. Por ello el proceso es el siguiente:

1. El usuario demandante mantiene una copia local de la ontología y elige en ella la categoría que mejor se adapte al servicio que está buscando.
2. Como se está usando un DAG, si la categoría seleccionada tiene varias categorías padre, o bien se escoge un camino desde la raíz o bien un subgrafo o fragmento de la ontología que contiene todos los caminos desde la raíz hasta la categoría elegida. Es una decisión de diseño cuál de las alternativas tomar.
3. El camino o subgrafo de categorías escogido es enviado dentro de un mensaje de demanda de servicio al resto de usuarios de la red (multicast).
4. Cuando otro usuario diferente al demandante recibe un mensaje de este tipo, *matcheará* el camino o subgrafo recibido con el de los servicios que él ofrece en su ontología.
5. Si se produce algún *matching*, el ofertante enviará una respuesta de éxito al demandante como resultado de la búsqueda.

No obstante, en el momento de comparar las rutas de categorías de los servicios ofertados con la ruta recibida pueden surgir problemas de ambigüedad cuando la coincidencia no es exacta sino parcial o similar. Estas son aquellas situaciones:

- La ruta solicitada es más genérica que la del servicio ofertado. En este caso, no se considerará como *matching* si el grado de generalización es demasiado alto comparado con lo particular que sea el servicio.
- La ruta solicitada es más particular que la del servicio ofertado. Si esto ocurre, se considerará *matching* al igual que antes en función de la cercanía de los nodos. No obstante, al solicitarse algo más particular que lo ofertado se preguntará al usuario ofertante para una confirmación final.
- Las rutas apuntan a la misma categoría pero hay diferencias entre ambas rutas debido a versiones diferentes de la ontología. Para lidiar con este problema conviene que el comportamiento de la combinación de ontologías sea lo más

ideal posible para que los nombres de las categorías y su clasificación sean lo más parecido entre los nodos de la red.

Otro aspecto importante en la búsqueda de servicios es el siguiente: ¿Qué ocurre si el demandante no encuentra en su ontología una categoría que se adapte a sus necesidades? La existencia de categorías depende de los usuarios que ofrecen servicios, ya que cuando necesitan añadir un nuevo nodo a la ontología tras la posterior difusión de la misma, se actualizará la del resto de nodos.

Para responder a una búsqueda de servicio, el par de la red tiene que usar un algoritmo de *matching* con el subgrafo de la petición y la ontología local. Este algoritmo podría incluir el reseteo de los TTL de las aristas y así permanecen aquellas relaciones entre categorías de la ontología conforme a lo que los usuarios buscan.

Ahora bien, del mismo modo que para la oferta, se expondrán tres casos de uso para describir la búsqueda de servicios.

5.5.3.1 Motivando el algoritmo de *matcheo*

Supondremos que el que solicita el servicio envía un camino y no un subgrafo.

Caso de uso 1:

En el primer caso, el demandante quiere buscar clases de inglés en el banco de tiempo, y para ello selecciona la siguiente ruta de categorías en su versión local de la ontología.

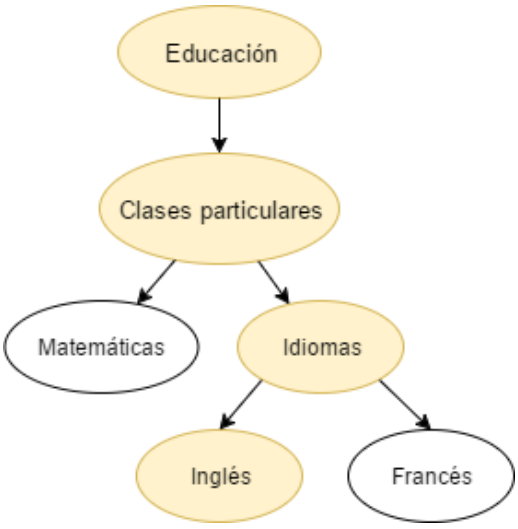

Figura 5.20: Demanda de servicios caso de uso 1 demandante

Se genera un mensaje que contiene una secuencia de las categorías seleccionadas y se propaga al resto de nodos de la red. En consecuencia uno de los nodos que recibe esta petición responde con un resultado pues éste tiene registrado entre sus servicios ofertados un servicio de clases de inglés clasificado como se indica en la versión de su ontología:

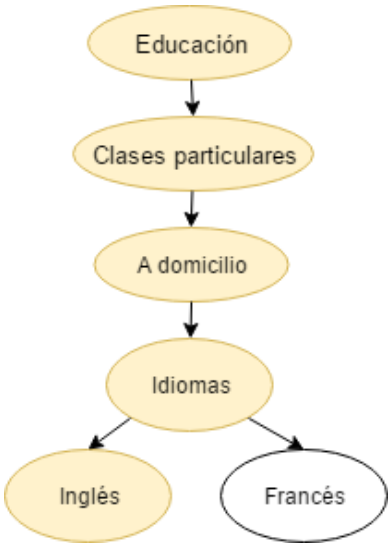

Figura 5.21: Demanda de servicios caso de uso 1 ofertante

Aunque la coincidencia de las dos rutas (la de solicitud y la de oferta) no es exacta, se considera como tal pues el servicio ofertado es un caso particular del servicio demandado: en este caso el grado de particularización no es elevado.

Caso de uso 2:

En el segundo de los casos, suponemos un usuario demandante que necesita un servicio de "paseo de perros" y por ello elige en su ontología local la siguiente ruta de categorías.

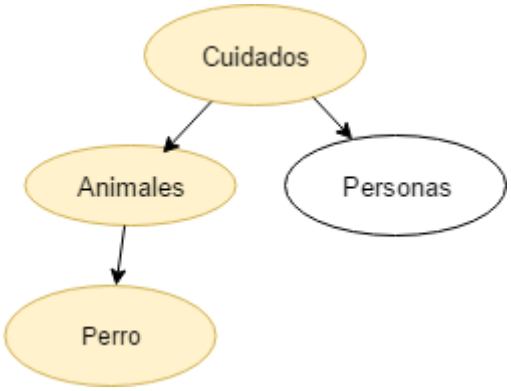

Figura 5.22: Demanda de servicios caso de uso 2 demandante

Tras ello, el mensaje con la ruta de categorías se difunde por la red a la espera de que algún nodo de la red responda con el servicio solicitado.

Uno de los nodos a los que les llega esta petición mantiene una ontología con el siguiente estado, y ofrece servicios de "cuidado de perros" pero no de paseo sino de baño:

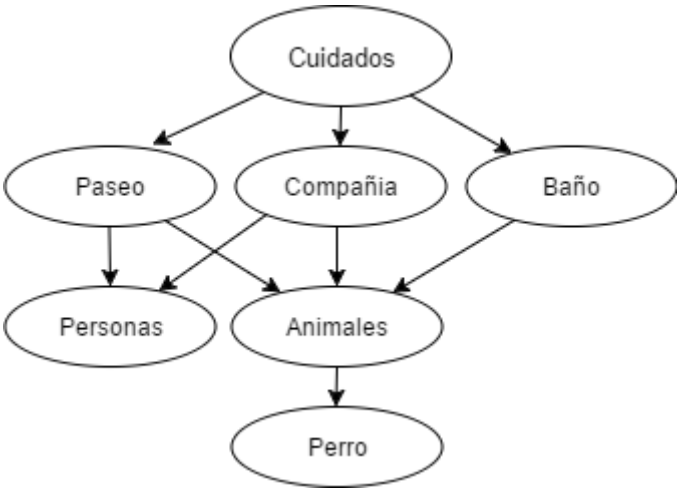

Figura 5.23: Demanda de servicios caso de uso 2 ofertante

Como en la petición del servicio no se especificaba el tipo de cuidado, el nodo del ofertante responde como éxito al haber encontrado coincidencias en la ruta de categorías recibida. El demandante recibirá como resultado de la búsqueda un servicio de baño de perros el cual no está entre los que él esperaba. Esto es debido al nivel de generalización que había en el mensaje de petición: el demandante indicó sus preferencias en un nivel demasiado alto. Con esto puede verse que cuantos más niveles de categorías tenga la petición, más acertados serán los resultados y por consiguiente en menor cantidad. En realidad, es el mismo comportamiento de búsqueda que puede haber en cualquier sistema de oferta y demanda.

Caso de uso 3:

En este caso, se describe el suceso de solicitar un servicio más específico que el ofertado por otro usuario. En primer lugar el demandante busca un servicio de clases de francés y elige la siguiente ruta según su ontología:

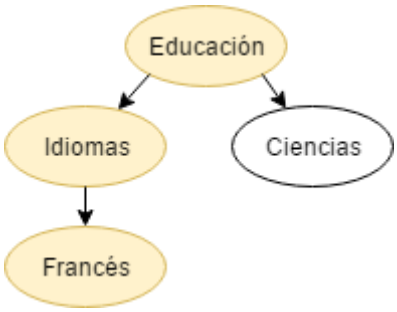

Figura 5.24: Demanda de servicios caso de uso 3 demandante

Por otro lado, otro usuario de la red mantiene una ontología con la siguiente estructura y ofrece un servicio de clases de inglés con la ruta indicada en la misma:

Figura 5.25: Demanda de servicios caso de uso 3 ofertante

Al ofrecerse un servicio clasificado en una categoría más general, aunque haya coincidencias en la ruta recibida, no es seguro que lo ofertado sea lo buscado por el usuario, como ocurre en este ejemplo. Es por eso que para no eliminar los casos en los que sí se debería responder, se opta por preguntar al ofertante para responder con un resultado de la búsqueda o no.

5.5.4 Infraestructura de red

5.5.4.1 Red P2P más adecuada

Una de las elecciones más importantes que se haga es la elección de la red distribuida sobre la que se construirá el sistema. Existen varios tipos de redes P2P, por lo que a continuación se resume la diferencia que existe entre ellas en función de su estructura:

- **No estructuradas:** la topología de la red depende de los nodos que haya actualmente conectados. Cada nodo almacena su propio contenido y mantiene una lista de sus vecinos. Cuando un nodo se une a la red le basta con conocer a otro nodo para estar dentro. En las primeras versiones de las redes no estructuradas la búsqueda de recursos e intercambio de información se realizaba mediante inundación, a través de la red, de mensajes con un tiempo de vida donde no se garantizaba encontrar el recurso solicitado aunque existiera. Hoy en día estas redes P2P han introducido una noción de jerarquía entre nodos, por ejemplo los supernodos en Gnutella.
- **Estructuradas**: la topología de la red permanece fija independientemente de los nodos que estén conectados. Para conectar un nodo se sigue un determinado procedimiento para ubicarlo en la red conforme a la topología fija, de este modo cada *peer* queda indexado de alguna manera. Si existe un recurso en la red se garantiza ser encontrado cuando sea solicitado. En general, en los sistemas P2P estructurados toda la información está distribuida por la red, es decir, el contenido creado por un determinado nodo no está necesariamente almacenado única y localmente por él.

En nuestro caso, la red que mejor se adapta a la parte de oferta y demanda de servicios de nuestra aplicación es la **no estructurada**. El motivo principal, a parte del bajo coste de mantenimiento frente a las estructuradas, es la **autonomía a nivel de nodo**. Por un lado esta autonomía es necesaria pues cada usuario de la aplicación almacenará él mismo la información (o al menos solo él tendrá el control sobre ella) de los servicios que ofrece y debe poder consultar o eliminar dichos datos cuando lo desee. Por otro lado, este tipo de redes son muy robustas debido a su alta tolerancia al *churn* (conexión

y desconexión de un gran número de nodos de manera frecuente). Esto es debido a que todos los peers juegan el mismo papel.

Debido al diseño previo que se hizo sobre el manejo de la información, en base a múltiples ficheros distribuidos por toda la red mediante una DHT, se eligió un framework estructurado como FreePastry para la parte de transacciones de nuestra aplicación. Por tanto, la aplicación usaría dos tipos de red P2P, una estructurada para la parte de transacciones y una parte no estructurada para la parte de oferta y demanda de servicios. Se ha hecho un estudio muy breve de la posibilidad de construir las dos partes de la aplicación sobre una jerarquía de redes estructuradas pero parece que el resultado sería demasiado complejo y rebuscado.

Una posible implementación de esta red no estructurada que se adapta muy bien a las necesidades es Gnutella [87], uno de los primeros grandes sistemas distribuidos desarrollado en el 2000. Las ventajas que Gnutella ofrece principalmente son:

- Sencillez y generalización: existen numerosas implementaciones debido a lo genérico que es su protocolo.
- Total descentralización.
- Organización dinámica: Gnutella ofrece una capa *overlay* con nodos auto-organizados, de tal modo que pueden entrar y salir de la red en cualquier momento.

Existen otros protocolos para redes no estructuradas pero menos extendidos como por ejemplo el protocolo Gossip, para el cual no existen tantas implementaciones. Asimismo, redes como eDonkey o FastTrack también funcionan de manera descentralizada y no estructurada pero sin embargo han quedado prácticamente en desuso en los últimos años.

5.5.4.2 Requisitos sobre la proximidad geográfica de pares

Por otro lado, en un sistema como el banco de tiempo, el comportamiento esperado es que los usuarios interactúen por lo general con los usuarios geográficamente más próximos a ellos (mismo barrio, misma ciudad, etc.). Es por eso que lo ideal sería contar con una red distribuida en la que los nodos estén dispuestos en relación a su localización física. Los trabajos sobre medidas de proximidad en redes P2P conciernen, en general, a la proximidad en la red (*network proximity*), ya que buscan aumentar la eficiencia de las búsquedas. Esta noción de proximidad no suele coincidir con la proximidad geográfica que se necesita en nuestra aplicación. Dicho problema puede tratarse de diferentes maneras:

- Una posible solución consiste en solicitar a los usuarios datos exactos de su localización. Esto supone un problema de privacidad aunque se puede mitigar exigiendo solamente información más general como ciudad o provincia.
- Se puede abordar el problema es construir una red no estructurada que contenga *super-peers* organizados por región. Se necesitaría de algún modo una capa *overlay* reorganizándose constantemente en función de la proximidad geográfica entre los nodos.

- Otra posible alternativa que se pensó fue construir el sistema usando tecnologías como WIFI-Direct [88] o LTE-Direct [89]. De manera simplificada son métodos para conectar varios dispositivos sin necesidad de un punto de acceso intermedio. De este modo, la información sobre la proximidad geográfica está siempre disponible en la red subyacente[2] y por eso son tecnologías que podrían encajar en una red P2P como la que se está buscando. No obstante todavía no están lo suficientemente extendidas ni desarrolladas.

5.5.4.3 Conexión y desconexión de nodos

Volviendo a uno de los problemas que surgieron durante la especificación de la oferta y demanda de servicios, se necesita definir qué hacer cuando un usuario *A* se desconecta y otro usuario *B* busca un servicio de los que el primero oferta pero no recibe respuesta pues *A* está fuera de la red y sus servicios no están visibles. Es una situación esperada pues un usuario no va a mantener la aplicación 24h conectada a la red por lo que habrá que encontrar solución. Una solución que se pensó es la siguiente:

Para que el usuario que solicita un servicio reciba respuesta de un usuario desconectado es necesario que la información de los servicios permanezca en la red. De este modo cuando un usuario ofertante abandone la red temporalmente deberá trasladar esa información a algún punto de la red para que siga estando disponible. Inversamente, esta información deberá regresar al nodo origen cuando éste vuelva a conectarse.
Para que esto pueda tener lugar de una manera controlada, no serviría trasladar la información a otros nodos con el mismo papel en la red (otros usuarios), ya que estos podrían desconectarse también por lo que de nuevo la información de los servicios de varios usuarios debería enviarse a terceros, y así sucesivamente si éstos también abandonan la red. Esta solución implicaría un desequilibrio de almacenamiento en el sistema, por lo tanto una solución más acertada es usar una red distribuida con supernodos (nodos diferentes a los de los usuarios) cuyo papel es almacenar la información de los nodos que se desconectan temporalmente del sistema e incluso almacenarla también cuando estén en línea.

5.5.4.4 TTL de los servicios

Para no acumular servicios de usuarios que han abandonado la aplicación, los servicios deben tener un tiempo de vida como ya se adelantó en el apartado de eliminación de servicios.

5.6 Conclusiones del diseño

Una vez realizado el estudio de la solución expuesta en la idea inicial llega el momento de poner en claro algunos puntos importantes definidos o que faltan por definir de cara a una futura implementación:

[2] El alcance de WiFi-Direct es de hasta 100 metros (con línea de vista directa) y de LTE-Direct es de hasta 500 metros.

Ontología de categorías

- Cada par de la red tendrá una ontología propia.
- La ontología será representada mediante un DAG en el que cada nodo de éste es una categoría formada por un conjunto de nombres que la representan. Hay una versión inicial de la ontología que se recibe al inicio de la aplicación en la cual una parte es permanente (probablemente sea todo el conjunto inicial aunque podría ser un subconjunto).
- Los nodos del DAG, y posiblemente también las aristas del DAG, salvo la parte permanente del mismo, tienen un tiempo de vida asignado cuando se crean (TTL). Al expirar este tiempo de vida, se borran del DAG local (salvo en el caso de un nodo que contiene un servicio cuyo propio TTL no ha expirado todavía). Los detalles de la asignación y gestión de los tiempos de vida quedan pendientes de ser definidos en función de los resultados de simulación.
- Cuando una categoría es eliminada los nodos hijos que quedan sin padre pasan a ser hijo de los padres de la categoría eliminada.
- La periodicidad del envío de la ontología a otros usuarios para el intercambio queda pendiente de ser definido en futuros procesos de simulación.
- Un usuario puede añadir, eliminar y alterar las relaciones en su ontología local.

Oferta de servicios

- La publicación se realiza de manera local, es decir, no se almacena esta información de manera distribuida por la red, en todo caso en un supernodo.
- Al ofrecer un servicio se resetea el tiempo de vida de las categorías implicadas.
- Cuando llega una petición con un camino o subgrafo de la ontología de otro usuario, se realizará un *match* para buscar emparejamientos con los servicios ofertados y también un *merge* con ese camino o subgrafo para actualizar la ontología local. Si se encuentra un match, también hay que actualizar los TTLs implicados.
- Un usuario puede eliminar los servicios que ofrece.
- Los servicios tienen un tiempo de vida, cuya asignación y gestión del serán definidos en función de los resultados de simulaciones que quedan por hacer. Cuando expire un servicio será eliminado. El usuario podrá renovar la oferta del servicio y con ello se reseteará su tiempo de vida y el de las categorías implicadas también.

Demanda de servicios

- El demandante seleccionará una categoría de su ontología local para buscar servicios de ese tipo. Si el demandante considera que su ontología local no tiene una categoría adecuada, puede añadir uno o más nodos y una o más aristas.
- La decisión sobre qué enviar en la petición aún no se ha tomado entre estas tres posibilidades:
    - Un camino desde la raíz hasta la categoría de servicios buscada.
    - El subgrafo de la ontología que contiene todos los posibles caminos desde la raíz hasta la categoría buscada, ya que se utiliza un DAG.
    - Dar la posibilidad al usuario demandante de limitar el subgrafo enviado, donde la limitación máxima es de un solo camino desde la raíz.
- El TTL de las categorías usadas en la solicitud se reseteará en la ontología local del demandante. Se debe investigar también la posibilidad de resetear los TTLs

de estas categorías en las ontologías de todos los pares por los que pasa la solicitud.

6 Implementación

En este capítulo se hablará de las implementaciones que se han hecho a lo largo del proyecto. Entre ellas se incluyen tomas de contacto con las tecnologías escogidas e implementaciones de demostración de módulos concretos del banco de tiempo.

6.1 Framework P2P elegido para el subsistema de transacciones

Para las implementaciones que se han hecho del banco de tiempo se necesita un framework o herramienta específico para construir una red P2P. Aquí se detalla el proceso seguido para elegirlo. En primer lugar la tecnología que elijamos necesita cumplir la mayoría de las siguientes premisas:

- Debe poder adaptarse a nuestro proyecto. Aunque estos frameworks sirvan para sistemas distribuidos es posible que el motivo por el que se especificaron no sea para el uso que le daremos nosotros.
- Debe tener un API o interfaz para poder construir sobre ella una aplicación sin entrar a la implementación del protocolo. Y ha de estar bien documentada para facilitar la implementación ya que no se goza de demasiado tiempo para invertir gran parte de éste en aprendizaje e investigación.
- Debe permitir almacenamiento distribuido. Por ejemplo mediante DHTs.
- Ser totalmente descentralizada. Si el objetivo del proyecto es construir una red P2P sin dependencia de un servidor, cuanto mayor sea el grado de descentralización mejor.

Nuestra aplicación va a necesitar almacenar información de cada usuario así como de las transacciones que se hagan, y una solución puede ser usar tablas hash distribuidas. Las DHTs proporcionan un coste de almacenamiento y acceso a los datos de $O(logN)$ en el mayor de los casos (siendo N el número de nodos de la red), y además ofrece replica de los datos. Las implementaciones de DHTs que más nos han llamado la atención han sido FreePastry y TomP2P ya que ambos cuentan con un sitio web propio con una amplia documentación fácil de acceder. A parte, su interfaz Java está diseñada de tal manera que el programador no tiene que preocuparse de la implementación interna y de bajo nivel de la red.

Por otro lado, nuestro sistema guarda registro de las transacciones de los usuarios del banco de tiempo, información que no debe ser alterada una vez guardada en la red. Para ello podríamos pensar en usar cadenas de bloques (Blockchain) ya que en ellas, los datos no se pueden modificar una vez almacenados. Sin embargo, si se quiere construir una red descentralizada usando blockchain es necesario que en cada nodo se tenga una copia de la cadena de bloques y que existan nodos mineros que se ofrezcan para realizar las verificaciones de las transacciones antes de almacenarse. Esto supone un gran coste computacional y energético, por lo que en nuestro proyecto no creemos que vaya a encajar como alternativa viable actualmente. Aunque como solución a este problema se podría construir la aplicación sobre algunas de las redes ya existentes sobre blockchain

como BitCoin [49] o Ethereum [90] pagando en cada transacción la cantidad mínima, ya que se exige una comisión para los mineros encargados de procesarlas.

Como se puede ver, existen muchas alternativas de implementación. Como no podemos realizar pruebas con todas y cada una de ellas hemos decidido estudiar solo aquellas que ofrecen una interfaz bien documentada y que creemos que mejor se vayan a adaptar a nuestro proyecto.

Después de investigar las posibilidades de uno y otro, creemos que las tecnologías que mejor se pueden adaptar a nuestras circunstancias actuales son FreePastry y TomP2P. Las motivaciones principales han sido:

- El lenguaje de programación: Java. Es conocido por los integrantes del proyecto por lo que no requiere invertir tiempo en aprender nuevos lenguajes.
- Almacenamiento mediante DHT ya implementado.
- Documentación y tutoriales.
- Permiten realizar pruebas de manera sencilla.

Como debemos elegir una sola de las dos alternativas, nos hemos decantado al final por FreePastry. La razón ha sido que FreePastry es un proyecto que ha tenido más apoyo y trabajo que TomP2P (y es que el proyecto TomP2P en los últimos años ha estado parado) y es la implementación de un protocolo muy probado como Pastry (una de las cuatro principales propuestas de DHT junto a Chord, CAN y Tapestry [91]).

En resumen, el framework sobre el que se implementará es FreePastry, aunque de cara a futuros desarrollos sería interesante probar otras tecnologías como Blockchain o TomP2P.

6.2 Framework elegido para el subsistema de oferta y demanda de servicios

Como ya se adelantó en el capítulo del subsistema de servicios, el tipo de red P2P que mejor se adapta en función de cómo se ha diseñado dicho subsistema, son las redes no estructuradas. En la sección 5.5.4 se habló sobre que una buena elección podría ser Gnutella., una red no estructurada muy probada en los últimos año que ha evolucionado a lo largo de ese tiempo.

6.3 Otras tecnologías usadas

Para las implementaciones que se detallarán en los siguientes apartados se han utilizado varias herramientas destinadas tanto al desarrollo del código como para el control de versiones.

Principalmente, el IDE (entorno de desarrollo integrado) usado para el desarrollo del código y sus pruebas ha sido Eclipse [92] y el lenguaje utilizado ha sido Java, aunque

también se ha usado C++ en Visual Studio para otras cosas como se explicará más adelante.

Dos de las implementaciones hechas se han construido sobre el patrón MVC (Modelo-Vista-Controlador) y éstas han sido la implementación de pruebas con el framework elegido y la implementación de demostración del protocolo de pago. Este patrón de diseño tiene la característica de separar la lógica de la aplicación de la interfaz ofrecida al usuario. De este modo se consigue la ventaja de no tener casi dependencia entre conceptos de manera que un cambio en la interfaz de usuario no implique cambios en la lógica y viceversa. Así pues se definen tres entidades: el modelo (lógica y datos), vista (interacción con el usuario) y controlador (intermediario). Existen varias interpretaciones de este patrón por ejemplo en función de quién notifica a la vista los cambios en el modelo, pero en este caso el flujo es el indicado por la Figura 6.1.

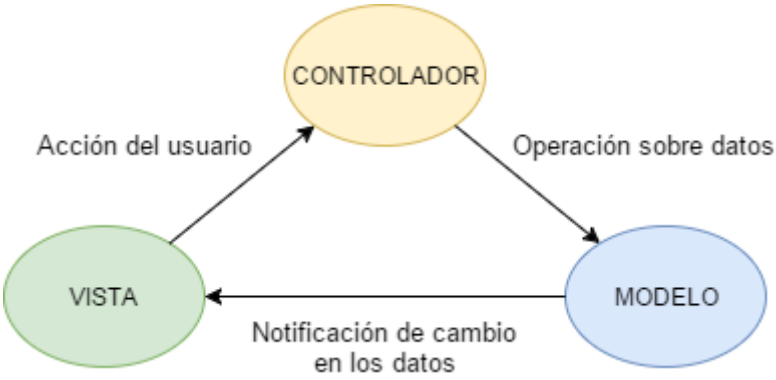

Figura 6.1: Esquema MVC.

Por otro lado, para el control de versiones se ha usado Git a través de la plataforma GitHub.

Git es uno de los controles de versiones más usados actualmente y es una buena solución cuando un software está siendo diseñado por varias personas y compuesto por numerosos ficheros. Git registra los cambios realizados así como quién los ha hecho. Destaca por su flujo de trabajo basado en ramas (no lineal), por su eficiencia y por su buen funcionamiento [93].

GitHub es una plataforma web que ofrece a sus usuarios repositorios para sus proyectos aplicando Git como control de versiones. Estos repositorios pueden ser públicos o privados en ellos se incluyen características tales como una Wiki, posibilidad de colaboraciones por parte de otros usuarios, seguimiento de la actividad, gestión de incidencias o problemas (*Issues*), entre otras [94].

6.4 Implementación de pruebas

Como ya se ha dicho anteriormente, el frameworks P2P que iba a ser usado es FreePastry [42]. Para poder entender mejor su funcionamiento y conocer las interfaces para los desarrolladores, se decidió construir un programa sencillo sobre este framework como primer contacto para poder implementar posteriormente ciertas funcionalidades del banco de tiempo con el mismo.

Se ha construido con ayuda de su documentación y tutoriales [95] una pequeña aplicación para probar la comunicación y uso de la DHT entre nodos de una red Pastry.

6.4.1 Qué hace el programa

Es un programa muy sencillo que cuenta con las siguientes funcionalidades:

- Cada ejecución del programa permite conectarse como un nodo a una red Pastry.
- Permite enviar un mensaje (String) a todos los vecinos del nodo.
- Permite almacenar y cargar contenido en la DHT:
    - Se almacena un String como contenido con un "put" en la DHT y la clave con la que se almacenó se envía a todos los nodos vecinos.
    - Con las claves de los contenidos que hemos guardado y las claves que nos han enviado nuestros nodos vecinos podemos hacer un "get" de todo ese contenido asociado en la DHT.

6.4.2 Requisitos previos

Para poder ejecutar el programa es necesario incluir los siguientes jars:

- FreePastry-2.1.jar [96].
- xmlpull_1_1_3_4a.jar [97].
- xpp3-1.1.3.4d_b2.jar [98].

6.4.3 Ejecución de ejemplo

Una vez que se hayan incluido los jar previos ya se puede ejecutar la aplicación:

1. En primer lugar se necesita levantar el primer nodo de la red para que el resto de nuevos nodos puedan unirse a ella. Para ello ejecutamos el programa y se hace lo siguiente:
    - Primero se requiere una IP válida (por ejemplo la que se tenga asignada por el router) sobre la que lanzar el nodo. Segundo, debemos indicar un puerto de arranque y de enlace idéntico cómo se muestra en la Figura 6.2: Ejecución de ejemplo paso 1.

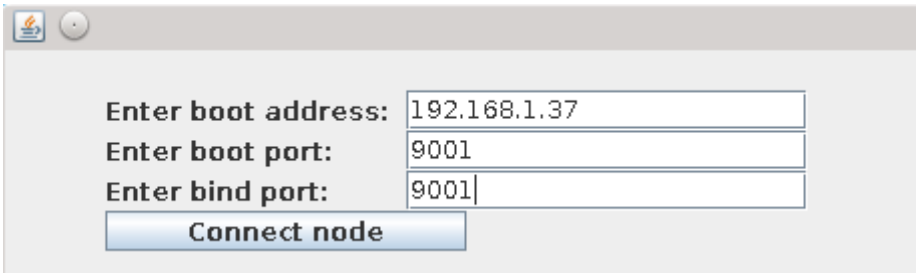

Figura 6.2: Ejecución de ejemplo paso 1

- En segundo lugar se pulsa el botón “Connect node” y aparecerá en el área de notificaciones el éxito (Figura 6.3) y en el indicador de ID de nodo veremos el ID que se le ha asignado a este primer nodo de la red (Figura 6.4).

Figura 6.3: Ejecución de ejemplo paso 1

Figura 6.4: Ejecución de ejemplo paso 1

2. Ahora que ya se tiene levantado el primer nodo ya se puede conectar un nuevo nodo a la red a partir de éste. Ejecutamos otra vez el programa y nos aparece una nueva ventana y repetimos el proceso de conexión anterior, pero esta vez con un puerto de enlace distinto. Y debemos poner la dirección y puerto de arranque del nodo de arranque que creamos antes (Figura 6.5: Ejecución de ejemplo paso 2). Tras pulsar el botón de conexión aparecerá el mensaje de éxito y el ID que se le ha asignado a este nuevo nodo (Figura 6.6: Ejecución de ejemplo paso 2).

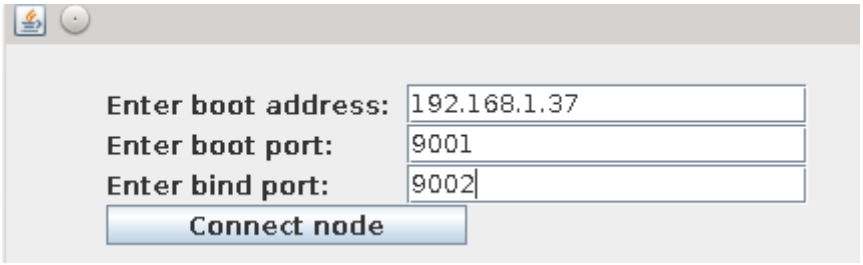

Figura 6.5: Ejecución de ejemplo paso 2

Figura 6.6: Ejecución de ejemplo paso 2

3. Ahora que ya hemos tenemos dos nodos en la red ya podemos enviar nuestro primer mensaje desde uno de ellos al otro. SI hubiera más nodos, el mensaje llegaría a todos los vecinos del que ha enviado el mensaje.
   - Desde una de las dos GUI (uno de los dos nodos) escribimos el mensaje que queremos enviar y pulsamos “Enviar” (Figura 6.7: Ejecución de ejemplo paso).

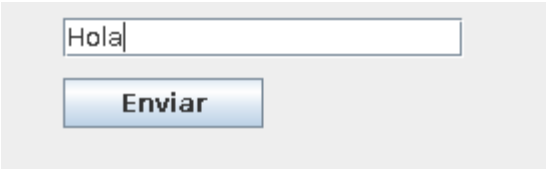

Figura 6.7: Ejecución de ejemplo paso 3

   - Comprobamos en el área de notificaciones del otro nodo que le ha llegado el mensaje (Figura 6.8: Ejecución de ejemplo paso 3).

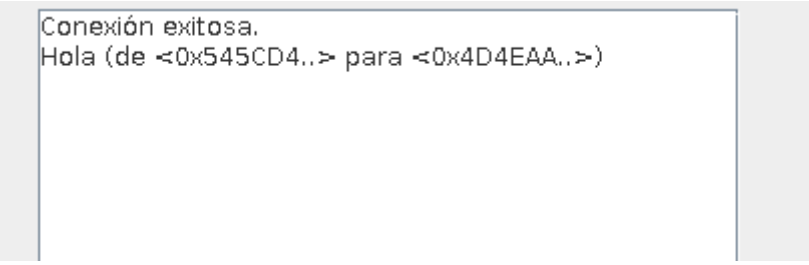

Figura 6.8: Ejecución de ejemplo paso 3

- Efectivamente el mensaje se ha enviado con éxito (Comprobamos el origen y el destino con los IDs de ambos nodos).

4. El siguiente paso es probar la funcionalidad del uso de la DHT:
   - Antes de nada, cabe señalar que en directorio donde se haya ejecutado el programa han aparecido dos nuevas carpetas. Estos directorios representan el almacenamiento Past asociado a cada nodo (storage<NodeID>) (Figura 6.9: Ejecución de ejemplo paso 4).

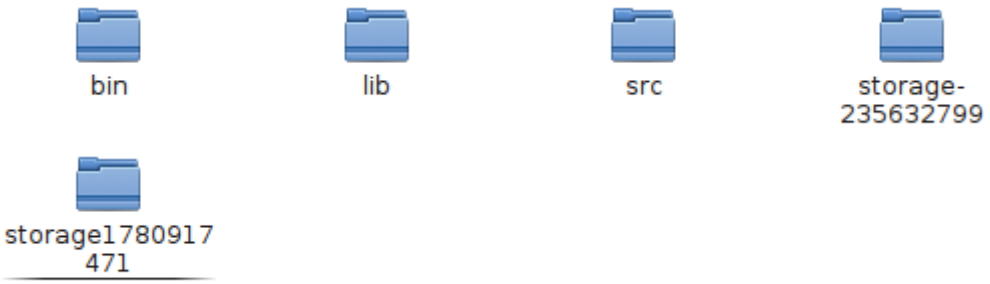

Figura 6.9: Ejecución de ejemplo paso 4

   - Primero es necesario explicar qué hace el programa:
     - Un nodo puede almacenar un String en la DHT, y cuando lo hace, la clave de acceso al contenido se envía a todos los nodos vecinos.
     - Cuando un nodo recibe una clave se le notifica en el área de notificaciones.
     - Un nodo puede consultar en la DHT el contenido asociado a todas las claves que ha recibido de otros nodos.
   - Ejemplo de funcionamiento:
     - Sin cerrar las ventanas creadas en el apartado anterior, escribimos en el campo de texto de encima del botón “Guardar contenido” el String que se quiere almacenar (Figura 6.10: Ejecución de ejemplo paso 4), y al pulsar dicho botón se almacena y se recibe una notificación en el propio nodo del número de réplicas del contenido que se han almacenado (Figura 6.11: Ejecución de ejemplo paso 4). Y en el otro nodo (el que no ha hecho el put) se recibirá una notificación de nueva clave recibida (Figura 6.12: Ejecución de ejemplo paso 4).

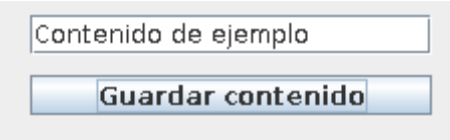

Figura 6.10: Ejecución de ejemplo paso 4

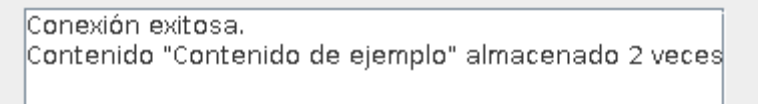

Figura 6.11: Ejecución de ejemplo paso 4

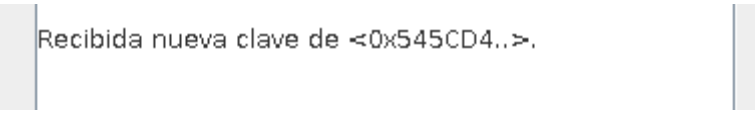

Figura 6.12: Ejecución de ejemplo paso 4

5. Ahora que ya tenemos un contenido almacenado que poder recuperar, desde el otro nodo (el que no hizo el *put*) pulsamos el botón "Recuperar contenido" y nos debería aparecer el *String* que se almacenó anteriormente (Figura 6.13: Ejecución de ejemplo paso 5).

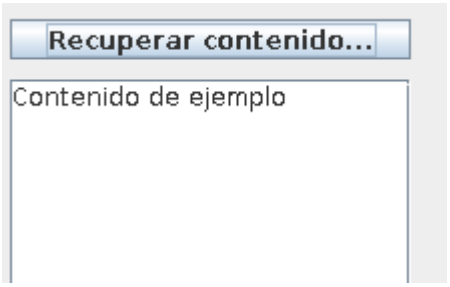

Figura 6.13: Ejecución de ejemplo paso 5

6.5 Implementación del protocolo de pago

6.5.1 Introducción

En vez de implementar todas las funcionalidades del Banco de Tiempo en su totalidad se van a implementar funcionalidades concretas para probar su funcionamiento y adaptabilidad a la tecnología P2P construida sobre una tabla de hash distribuida (DHT). En este caso se va a implementar una demo del protocolo de pago, es decir, la parte en la que el usuario que ha recibido un servicio debe pagar al usuario que se lo ofreció los créditos acordados. Este protocolo es el núcleo de la parte de las transacciones, por lo que puede verse también como el núcleo del banco de tiempo. Como se ha visto en la sección 4.3.2 donde se presenta este protocolo, tiene bastante complejidad.

6.5.2 Objetivo

El objetivo de la demo es construir una aplicación que permite a dos nodos realizar un pago siguiendo el protocolo de pago definido y comprobar que se ha almacenado correctamente la información en la DHT.

6.5.3 Contexto

Para simular el protocolo de pago es necesario que se prepare un contexto en el que la factura (***Bill***) del servicio realizado ya se haya generado, para que el deudor (***Debtor***) pueda empezar a iniciar el pago cuando desee y se tenga información para rellenar los ficheros pertinentes. Es decir, se supone que previamente un usuario acreedor (***Creditor***) dio un servicio a otro usuario y ahora este último debe iniciar el proceso en el que se actualiza el crédito de horas de ambos usuarios del banco de tiempo (el de uno aumentará y el del otro disminuirá).

6.5.4 Flujo de ejecución

Los pasos de este programa de demostración son los siguientes:

1. Inicialmente se creará un nodo en la red de FreePastry para poder almacenar en él el contenido previo al pago y para que los dos usuarios de éste se puedan conectar.

2. Se creará un perfil público para cada uno de los usuarios y se almacenarán en la DHT. La aplicación guardará los hashes.

3. Se utilizará la información de los perfiles públicos para generar una factura (*bill*) con datos inventados (el objetivo es probar el funcionamiento del protocolo), se almacenará en la DHT y la aplicación guardará el hash.

4. Asimismo se creará un asiento contable (***AccountLedgerEntry***) por cada usuario y se almacenarán en la DHT tras guardarse sus correspondientes hashes.

5. Posteriormente se crearán dos nodos en la red de FreePastry los cuales se conectaran a ella mediante el primer nodo que se creó en el primer punto.

6. Se lanzarán dos procesos distintos (uno por cada usuario) simulando dos sesiones de usuario iniciadas.
    a. Ventana del deudor (*Debtor*): este usuario podrá ver el pago disponible que tiene y podrá iniciarlo.
    b. Ventana del acreedor (*Creditor*): a este usuario le llegará una notificación de inicio de pago por parte del deudor.

7. Una vez que el deudor ha iniciado el pago, tendrá lugar la primera fase del protocolo de pago. El Creditor recibirá un mensaje de notificación y ya tendrá disponible la opción para continuar con el proceso.

8. Una vez que se lleven a cabo todas las fases del pago definidas en su protocolo, ya se habrán almacenado los ficheros persistentes en la DHT y podrán tener acceso a ellos ambas partes del pago para comprobar que se almacenaron correctamente.

6.5.5 Interfaz gráfica

Cabe destacar que el diseño de la interfaz no ha sido lo primordial en esta demo, el principal objetivo es tomar el contacto con el framework FreePastry e implementar una primera versión del protocolo de pago, por lo que no nos hemos centrado en programar una interfaz bien diseñada.

Para la interfaz gráfica son necesarios los siguientes elementos:

- Sección para notificaciones.
- Sección con las transacciones pendientes:
    - Por cada transacción pendiente que haya se indicará el estado en el que se encuentra el proceso, y dependiendo de si el usuario es deudor, hay opciones activas o no.
- Sección para logs de ejecución: El objetivo de este componente es tener seguimiento del almacenamiento o carga de los ficheros de la DHT, entre otras operaciones.
- Sección para rellenar los *feedback*: será un cuadro de diálogo que aparecerá cuando sea necesario rellenar una opinión acerca del servicio recibido o dado.

6.5.6 Ejecución de ejemplo

Para poder ejecutar el programa son necesarios los tres mismos *jars* mencionados en la implementación de prueba anterior.

En primer lugar es necesario tener en cuenta algunas consideraciones antes de ejecutar el código. Para que un nodo pueda ser añadido a la red, le hace falta tener una IP y un puerto válidos sobre el que trabajar. Como la demo levanta tres nodos en la misma máquina se deberá indicar al inicio de la aplicación una IP válida y tres puertos distintos.

A continuación se detallará un ejemplo de uso de la demo:

1. Lanzamiento del programa: para ello solo habrá que realizar una única llamada al método principal de la demo:
    a. En la salida de consola al inicio del programa se pueden ver los logs que indican que se ha lanzado un nodo de arranque principal, que se han almacenado los ficheros iniciales necesarios, y que se han lanzado los dos nodos correspondientes al acreedor y al deudor.

   b. Aparecerán dos ventanas, una para el acreedor y otra para el deudor. Cada una de las secciones de la interfaz gráfica definidas anteriormente aparecen rellenas con la información cargada de los ficheros almacenados previamente en la DHT: Se habrá cargado una factura que aparecerá en la sección de transacciones y las operaciones internas habrán reflejado su resultado y ejecución en la sección de logs.
   c. En este caso la diferencia entre la ventana de un usuario y otro es que el acreedor al abrir una factura no puede hacer nada más que verla, mientras que el deudor tiene presente una opción para iniciar pago.
2. Una vez cargadas las dos ventanas, el siguiente paso es interactuar con la interfaz del deudor para iniciar el pago.
   a. Se selecciona en la ventana del deudor la factura que hay en la lista de transacciones y tras ello aparecerá un diálogo en el que se mostrará información de la factura y una opción preguntando si el usuario quiere iniciar ahora el pago del servicio.
   b. Tras pulsar en la opción de iniciar pago aparecerá un formulario para introducir el *feedback* del deudor acerca del acreedor. Este *feedback* consiste en un comentario y en un número del 1 al 5 indicando su grado de satisfacción con el servicio recibido.
   c. Ahora que se ha contestado el formulario del *feedback*, la aplicación iniciará la primera fase del protocolo de pago. A medida que se completa esta primera fase se irán mostrando los logs de la carga y almacenamiento de ficheros en la DHT. Al final de esta etapa se enviará una notificación al acreedor con información para que pueda proseguir con el pago.
3. El acreedor recibe una notificación por parte del deudor en la mitad de la fase uno del pago y dicha recepción será registrada en el log y en la lista de notificaciones de la interfaz:
   a. Tras seleccionar la notificación recién recibida, el acreedor verá un mensaje indicando que el pago ha sido iniciado por el deudor y se le pregunta si desea proceder con la validación de los ficheros parciales generados en la primera fase y empezar con la siguiente fase.
   b. Si se eligió la opción de proceder con el pago, aparecerá un formulario de *feedback* idéntico que el del deudor pero en este caso será del acreedor acerca del deudor.
   c. Tras rellenar este formulario se validarán los ficheros creados en la primera fase por el deudor y se ejecutará la primera etapa de la fase dos del pago. Se mostrarán los logs de ejecución correspondientes en la sección de logs de la interfaz del acreedor. Asimismo se enviará una notificación al deudor para que éste ejecute la siguiente fase del pago.
4. El deudor recibe una notificación que será registrada tanto en el log como en la lista de notificaciones recibidas.
   a. Al abrir la notificación recibida, se mostrará un mensaje al deudor sobre si desea validar los ficheros generados por el acreedor en esta segunda fase y comenzar la fase tres del pago.
   b. Al confirmar el mensaje mostrado se ejecuta inmediatamente la validación y la fase tres del pago del deudor. Los logs de ejecución correspondientes quedarán registrados y una nueva notificación será enviada al acreedor.

5. El acreedor recibe la notificación del deudor del mismo modo que recibió la primera notificación en fases previas.
    a. El mensaje de esta notificación es para confirmar la validación de los ficheros parciales generados por el deudor en la tercera fase del pago y para confirmar el inicio de la última fase del pago.
    b. Tras finalizarse esta fase, el pago ya ha terminado y se envía una notificación de fin de pago al deudor.
6. El deudor recibe una notificación del acreedor del mismo modo que recibió la otra notificación en una fase previa.
    a. Al abrir la notificación se muestra un mensaje meramente informativo indicando que el pago finalizó con éxito.

6.5.7 Limitaciones encontradas en el framework

En este apartado se describirán las limitaciones en el desarrollo de la aplicación en el caso de seguir desarrollándose con el framework FreePastry.

No se permite editar un fichero una vez almacenado en la DHT: en realidad esta limitación iba a ser encontrada en la gran mayoría de los frameworks de redes P2P. Este hecho provoca que haya que rediseñar el modelo de datos ligeramente respecto a cómo estaba diseñado previamente.

Cuando se almacena un fichero en un sistema que utiliza una DHT se hace de manera distribuida y normalmente mediante réplicas del mismo en varios puntos de la red. Por consiguiente si se pretende editar ese mismo fichero habrá que editar también todas sus réplicas. Sin embargo, si en el momento de esta edición una de las réplicas del fichero se encuentra almacenada en un nodo desconectado, habrá inconsistencia cuando el nodo vuelva a conectarse a la red, es decir, habrá dos ficheros con la misma clave en la DHT pero diferente contenido.

Con esta limitación el editar un fichero conllevaría la creación de un nuevo fichero y diferente por separado y con una nueva clave en la DHT. Esto tendría implicaciones en el diseño de algunas funciones del sistema como la de editar los datos de la cuenta.

A causa de esto, en la implementación del protocolo de pago se tuvo que hacer un cambio en cuanto al modelo de datos ya definido. En este modelo se indicaba que el perfil del usuario mantenía la clave de la DHT del último elemento de las listas de AccountLedgerEntries, FAMs (Feedback About Me) y FBMs (Feedback By Me), sin embargo, como los ficheros no son editables en la DHT, no se puede modificar el fichero del perfil público con una nueva clave cada vez que se añada un elemento a cualquiera de estas listas, habría que crear otro fichero de perfil público y por consiguiente otra clave para éste. Por eso, en esta implementación de ejemplo, el puntero a la lista es al primer elemento con una clave DHT precalculada en el caso de lista vacía.

6.6 Implementación de la ontología

Desde un principio se quiso llegar a una implementación lo suficiente estable de la ontología de categorías para poder realizar las pruebas correspondientes de simulación y así establecer los primeros valores para los valores de TTL o periodicidad de intercambio, entre otros.

La primera implementación que se realizó se hizo en el lenguaje de programación C++ pues permitía un manejo más explícito mediante punteros sobre los nodos del grafo, aunque también se podía haber realizado en Java.

Se ha creado para ello una clase llamada *Node* que representa un nodo de la ontología cuyos atributos son el nombre de la categoría que representa y la lista de sus hijos. Tiene métodos tales como consultar nombre e hijos así como comprobar si tiene un hijo con un nombre dado o incluso añadir un nuevo hijo.

Se ha definido por otro lado un tipo de datos llamado *tEdge* (arista) el cual puede representar una arista de un grafo o un único vértice. Consta de tres campos, un identificador de la *cabeza* de la arista (hijo), un identificador de la *cola* de la arista (padre) y un *flag* que indica si la estructura de datos en cuestión es una arista o un único vértice, si es un vértice único entonces solo el campo de cola tendrá valor. Una lista de aristas de tipo *tEdge* es también una representación de un grafo, no necesariamente un DAG (en el sentido de que un grafo en realidad es un conjunto de aristas).

En un nivel superior, se ha usado esta clase *Node* para construir una clase llamada *Ontology* (la cual es en realidad un DAG) y consta de una lista de nodos raíz de tipo *Node* (no es un rooted-DAG) y ofrece una constructora que puede recibir una lista de aristas de tipo tEdge a partir de la cual se construirá el DAG, o dos listas de aristas de tipo tEdge, es decir, la operación *merge*. Como una lista de aristas de tipo tEdge no representa necesariamente un DAG y la unión de dos de ellas puede resultar siendo un grafo con ciclos, se ha optado por lanzar una excepción en el momento en el que se detecte un ciclo durante el proceso de construcción a partir de aristas.

El algoritmo seguido para la operación *merge consiste* en la construcción de un DAG a partir de una lista de aristas de tipo tEdge tal como se mostró anteriormente en los casos de ejemplo durante la fase de análisis. A continuación se muestra en pseudocódigo el constructor de Ontology que recibe dos listas de aristas de tipo *tEdge*:

```
List<Node> rootsList;   //atributo de Ontology. Lista de nodos raíz.

constructor Ontology(List<tEdge> listA, List<tEdge> listB) {
    listC = concat(listA, listB);

    edgesListToOntology(listC);
}

void edgesListToOntology(List<tEdge> edgesList) {
  if (!empty(edgesList)) {
        //Conjunto con las aristas ya incluidas en el grafo. Es
        necesario dado que puede haber aristas repetidas en la lista y
        así comprobar cuales se han leído ya y no repetir cómputo
        Set<tEdge> checkedEdges;
```

```
      //Diccionario con los nodos del DAG que se está construyendo.
      //La clave es el nombre del nodo y el valor es el nodo.
      HashMap<String, Node> nodes;

      //Conjunto formado por los identificadores de los nodos que
      actualmente son raíces en el DAG que se está construyendo. Al
      final del procesamiento, las raíces de la ontología serán los
      nodos con identificador en este conjunto.
      Set<String> possibleRootsNodes;

for (edge in edgesList) {
  if (!checkedEdges.contains(edge)) {
    if (edge es una arista) { //si no es un único vértice
      if (los dos nodos de edge están en el grafo actual
          pero no conectados de manera directa) {
        if (edge.cabeza tiene como hijo en el grafo actual a
              edge.cola) {
            throw Exception ciclo encontrado
        }
        else {
             Añadir en el grafo a edge.cola como hijo edge.cabeza.

             if (edge.cabeza está en el conjunto de posibles nodos
               raíz) {
                 Se elimina edge.cabeza del conjunto
                 possibleRootNodes.
             }
        }
      }
       else if (edge.cola no está en el grafo pero edge.cabeza sí){
        //Se crea un nuevo nodo con el  mismo identificador que
         edge.cola, se le añade como hijo el nodo edge.cabeza que
         hay en el grafo actual y se añade a la lista de posibles
         nodos raíz ya que no es hijo de momento de ningún otro
         nodo; y al diccionario de nodos ya insertados.

            Node newNodo(edge.cola.id);
            newNode.addChild(nodes.get(edge.cabeza.id));
            nodes.insert(edges.cola.id, newNodo);
            possibleRootNodes.insert(newNodo);
       }
       else if (edge.cola está en el grafo pero edge.cabeza no) {
          //Se crea un nuevo nodo con el mismo identificador que
         edge.cabeza, se añade como hijo al nodo edge.cola que hay
         en el grafo actual y se añade al diccionario de nodos ya
         insertados.

            Node newNodo(edge.cabeza.id);
            nodes.get(edge.cola.id).addChild(newNodo);
            nodes.insert(edges.cabeza.id, newNodo);
       }
       else { //Ninguno de los nodos de la arista está en el grafo
          //Se crean dos nuevos nodos con el mismo identificador
         que edge.cabeza y edge.cola. Se añade como hijo la cabeza a
         la cola y ambos se insertan en el diccionario de nodos ya
         insertados en el grafo. La cola se añade al conjunto de
         posibles nodos raíz ya que de momento no es hijo de ningún
         otro nodo.

             Node newCola(edge.cola.id);
```

```
                Node newCabeza(edge.cabeza.id);

                newCola.addChild(newCabeza);
                nodes.insert(edges.cola.id, newCola);
                nodes.insert(edges.cabeza.id, newCabeza);
                possibleRootNodes.insert(newCola);
            }
          }
          else { //si es un único vértice
            //no existe edge.vértice en el DAG
            if (nodes.get(edge.vértice.id) == null) {
                //Se crea un nuevo nodo con el mismo identificador y se
                añade a la lista de posibles nodos raíz ya que no guarda
                relación con ningún otro nodo.

                Node newNodo(edge.vértice.id);
                possibleRootNodes.insert(newNodo);
                //Se marca la arista recién leída como leída.
                checkedEdges.insert(edge);
            }
          }
        }
      }

      for (id in possibleRootsNodes) {
        //Se coge el id del posible nodo raíz, Se utiliza como clave en
        el diccionario nodes y el nodo devuelto se añade a la lista
        final de nodos raíz de la ontología.
              rootsList.add(nodes.get(id));
      }
    }
    else {
        //Una lista vacía de nodos raíz significa una Ontología o DAG
        vacío, es decir, sin nodos
        rootsList = emptyList();
    }
}
```

7 Validación

Como la mayor parte del proyecto ha girado en torno a la definición del subsistema de servicios del banco de tiempo (oferta, demanda y ontología de categorías), solo se ha podido realizar una primera implementación de la ontología y del algoritmo de *merge* de dos ontologías. En el caso de que en un futuro alguien se disponga a continuar con el trabajo y se llegue a una implementación con la que realizar pruebas y simulaciones, estos son los datos que habría que extraer:

- TTL de las categorías de la ontología.
- TTL de las aristas de la ontología si finalmente se decide añadir.
- TTL de los servicios publicados.
- Decidir qué es más adecuado al construir el mensaje de petición en la demanda de un servicio: un único camino desde la raíz hasta la categoría elegida en la ontología, o un subgrafo de ésta.

El método para sacar los valores de los TTL consistiría en realizar varias iteraciones sobre el mismo proceso de simulación a partir de unos valores iniciales, estudiar los resultados y establecer nuevos valores hasta encontrar unos que se adapten a las condiciones deseadas. Del mismo modo ocurriría para tomar la decisión sobre cómo formar el mensaje de solicitud de servicios, se deben realizar pruebas hasta comprobar qué alternativa es más idónea.

8 Trabajo futuro

Existe un gran número de posibles maneras de continuar con este proyecto en el futuro. A continuación se enumeran algunas de ellas:

- La red P2P ideal para el subsistema de oferta y demanda de servicios definido es una red no estructurada, sin embargo el resto de módulos de la aplicación definidos en el proyecto previo se pensaron para una red estructurada con DHT. Existen dos posibles caminos a seguir:
  - Estudiar una posible manera de usar los dos tipos de redes en la misma aplicación.
  - Redefinir el sistema para que utilice o bien solo una red estructurada o bien solo una red no estructurada.

- En el proyecto previo se propuso como trabajo futuro, y que no ha sido posible probar en éste, una simulación de una red P2P mediante OverSim o FreePastry's Simulator.

- Aunque se haya hecho ya una primera implementación de la ontología y su operación *merge* (en C++), sería interesante estudiar cómo implementarla usando el Java-SDK de Microsoft Azure [99], que ya tiene clases para grafos, DAGs, nodos de un DAG, etc.

- Incluir en el diseño del algoritmo *merge* las siguientes consideraciones ya tratadas en este documento:
  - Incluir *subsumption*, es decir, tener en cuenta relaciones de hiponimia e hiperonimia.
  - Eliminación de aristas redundantes.
  - Actualización de los TTLs.

- Realizar una primera implementación de todo el sistema de oferta y demanda sobre una red no estructurada (Gnutella es una buena opción como ya se adelantó) para probar su funcionamiento.

- Definir el resto de módulos pendientes de la aplicación: sistema de notificaciones y un sistema de reputación más sofisticado.

- Realizar la validación descrita en el capítulo anterior.

9 Conclusiones

9.1 Diferencias y novedades respecto al proyecto previo

Como ya se adelantó en la introducción, el proyecto aquí presentado es la continuación de otro proyecto previo, respecto al cual se indicó que podría haber algún solape inevitable con el trabajo realzado. Sin embargo, al final estos solapes han terminado siendo mínimos, y éstos han sido los estudios realizados sobre la tecnología P2P en el estado del arte. Este punto dentro del trabajo era totalmente necesario puesto que había que acercarse e introducirse a las tecnologías que se iban a usar. A pesar de ello, este proceso de investigación se realizó desde cero y de manera independiente.

El resto de aspectos comunes se encuentran en la especificación del banco de tiempo y todo lo que ella incluye (análisis, diseño e implementación) ya que la propuesta de este proyecto era una continuación del desarrollo de una aplicación y no se ha partido de cero para terminar de definirla. No obstante los puntos que se presentan en este documento sobre la definición del banco de tiempo son las correcciones que se le han hecho y las nuevas aportaciones propias de este proyecto.

Estas nuevas aportaciones se detallarán más adelante en un apartado específico para ello.

9.2 Objetivos cumplidos

Comparando los objetivos propuestos al inicio del curso y, enumerados en la introducción de este mismo documento, con los que verdaderamente se han logrado llevar a cabo podemos decir que sí se han cumplido todos ellos aunque algunos de ellos en mayor medida que otros.

En primer lugar, el estudio de las tecnologías P2P que se llevó a cabo sirvió para poder concebir los problemas planteados desde una visión descentralizada ya que la aplicación diseñada contaba con el problema añadido de que no iba a ser construida sobre un servidor central.

El siguiente objetivo perseguido fue el de realizar una corrección de la definición existente del banco del tiempo hecha por los alumnos del proyecto inicial. Se hicieron múltiples ajustes tanto en el análisis como en el diseño. Hubo cambios en la mayoría de los casos de uso que consideramos necesarios así como en el modelo de dominio de la aplicación. Otra importante modificación llevada a cabo fue la del protocolo de pago: el protocolo que permite a un usuario del banco de tiempo pagar las horas acordadas a otro usuario que le ha prestado un servicio determinado.

Respecto al objetivo de implementar alguna funcionalidad del banco de tiempo se consiguió realizar una pequeña versión de demostración del nuevo protocolo de pago definido. Sin embargo, no se le dedicó todo el tiempo deseado pues era más prioritario e

interesante tratar otros aspectos como el de buscar soluciones a otros problemas de una aplicación P2P de utilidad social (el último de los objetivos propuesto al inicio).

Finalmente, tras haber estudiado las tecnologías P2P y haber corregido el diseño del banco de tiempo previo, se decidió dejar temporalmente la implementación para definir uno de los módulos del banco de tiempo que no lo estaba y así encontrar alternativas P2P para aplicaciones similares. El módulo definido fue el de oferta y demanda de servicios. Con esta definición se puede decir que el objetivo de buscar soluciones generalizables a otras aplicaciones P2P se consiguió pues existen numerosos sistemas cuyo pilar es la publicación y búsqueda de productos o contenido, entre otros.

El equipo de trabajo inicialmente estaba formado por dos alumnos, pero uno de ellos tuvo que dejar el proyecto al inicio del mismo, por lo que todo el trabajo planificado recayó en el otro alumno. En consecuencia la velocidad con la que se avanzaba se redujo a la mitad. Este ha sido uno de los motivos por los que se tuvo que dejar de lado la implementación y es que hubiera sido muy interesante tener una buena implementación de la ontología de categorías con la que realizar simulaciones para decidir sobre el valor de sus parámetros y poder comprobar el funcionamiento de la función *merge*.

Aunque las implementaciones que se han realizado tanto del protocolo de pago como de la ontología sean actualmente versiones muy simplificadas, sí que se ha conseguido un acercamiento a cómo sería una construcción real de estas funcionalidades y con ello saber qué limitaciones tiene usar el framework elegido, y cómo debe funcionar y qué aspectos añadir al algoritmo *merge* de ontologías.

9.3 Aportaciones

La primera aportación de este proyecto ha sido añadir coherencia y completitud al proyecto inicial del banco de tiempo como se ha explicado en los objeticos cumplidos y a lo largo del documento. Los cambios han logrado plasmar de una manera más clara y concisa las especificaciones tanto del proyecto pasado como las que se han añadido aquí.

Siguiendo el hilo de las contribuciones que acompañan a los objetivos cumplidos, una de las aportaciones más importantes dentro de este proyecto ha sido el estudio y definición parcial del sistema de oferta y demanda de servicios. Se ha realizado una propuesta reutilizable para múltiples tipos de aplicaciones que se quieran construir sobre una red P2P y que utilicen la oferta y demanda de productos, servicios, contenido, etc. Esto se consigue gracias a la ontología dinámica de categorías que va adaptando su topología conforme a la actividad de los usuarios, de manera que mantendrá un conjunto de las categorías más populares y actuales. Actualmente no existe casi nada parecido [85].

Asimismo también se ha especificado una primera versión de una ontología de categorías para clasificar servicios (5.5.1.2), que puede ser utilizada en muchos otros contextos, no solo en el banco de tiempo o en aplicaciones descentralizadas.

En este punto cabe destacar la importancia de una buena definición, en la que es necesario realizar un estudio amplio, razón por la cual no hubo tiempo para la construcción de una aplicación completa o una versión avanzada de ésta. Sin embargo, sí que se ha comenzado una implementación de la ontología y la función *merge*, la cual es una buena base como trabajo futuro.

9.4 Conclusiones sobre el valor de las aplicaciones P2P

A lo largo del desarrollo del proyecto han ido apareciendo diversos obstáculos debido al tipo de aplicación que se estaba diseñando ya que existe un número inmenso de aplicaciones de carácter social pero muy pocas construidas enteramente sobre una red P2P sobre las que apoyarse.

Una de las razones principales para este hecho es la dificultad de encontrar un modelo de negocio viable para aplicaciones implementadas sobre redes P2P. Las grandes empresas del sector tecnológico no invierten apenas en la investigación de soluciones P2P pues con un sistema descentralizado no es obvio como generar beneficios, en particular, es difícil introducir anuncios y es difícil monitorizar el comportamiento de los usuarios, o bien con el fin de cobrar directamente los servicios usados, o bien con el fin de rentabilizar la información recogida sobre sus actividades en el sistema. Por otro lado, es difícil controlar el comportamiento de los usuarios para evitar que usen el sistema para actividades delictivas. Del mismo modo, puesto que los programas de investigación de las autoridades públicas suelen reflejar las inquietudes de las grandes empresas, tampoco hay mucha inversión pública en la investigación de soluciones P2P.

Sin embargo, no es necesario que sean las empresas privadas las que fomenten y ayuden a proliferar el mundo de la tecnología P2P. Las autoridades públicas también podrían participar en esta causa por ejemplo creando aplicaciones útiles para los ciudadanos mediante este tipo de redes. Un ejemplo de iniciativa al desarrollo de software por parte de un gobierno, aunque en este caso no se trata de una aplicación P2P, es la ciudad de Copenhague donde se ha desarrollado una aplicación para conocer el tráfico de ciclistas debido al alto número que hay de éstos [100]. Si existen iniciativas para este tipo de sistemas de utilidad social, por qué no invertir en soluciones descentralizadas con el mismo propósito y sacar provecho de sus ventajas.

10 Conclusions

10.1 Differences and novelties regarding the past project

The project presented here is the continuation of a previous project as it was indicated at the introduction of this document, there it was mentioned too that it might be some overlaps between both works. Nevertheless, at the end these overlaps turned out to be minimal, and they have been the studies about P2P technology in the state of art chapter. This point inside of this work was totally necessary to make an approach and introduction to the technologies that were going to be used. Despite that, this research process was started from scratch and independently.

The rest of common aspects are in the time bank specification and all that it includes (analysis, design and implementation) since the proposal of this project was the continuation of the development of an application and we don't have started from scratch to finish defining it. However the points presented in this document about the time bank definition are the corrections of previous specification and the new contributions of this project.

These new contributions are described later in a specific section dedicated to it.

10.2 Achieved goals

Comparing the goals proposed at the beginning of this course and, listed in the introduction of this document, with the goals truly reached, we can tell that all of them have been fulfilled although some of them more than others.

First, the study of P2P technology that was carried out served to be able to conceive the problems raised from a decentralized point of view since the application was though not to be built over a centralized server.

The following pursued aim was to make a correction of the initial definition of the application made by the students of the past project. Multiple adjustments were made in both analysis and design. There were changes in most of the use cases as well as in the domain model. Another important modification was the payment protocol: the protocol that allows an user of the time bank to pay the agreed hours to another user who has provided him a service.

About the goal of implementing some functionality of the time bank, it was possible to perform a little demo version of the new defined payment protocol. However, it was not invested all the desired time because it was more priority and interesting to attend others aspects as finding solutions to other problems of a social utility P2P application (the last of the proposed goals at the beginning).

Finally, after having studied the P2P technologies and having corrected the previous time bank design, the decision of putting aside the implementation was taken to define one of the non-defined modules of the time bank and thus find P2P alternatives to similar applications. The chosen module was the publication and search of services subsystem. With this definition it can be said that the goal of finding generalizable solutions to other P2P application was achieved since there are a lot of systems which use the publication and search of products or content, among others.

The work team was formed by two students, but one of them had to leave the project at the beginning of it, so all of the planned work fell into the other student. Consequently the speed of the project was reduced to its half. This has been one of the reasons to having put aside the implementation and it would have been very interesting to have a nice version of the ontology of categories to make simulations and decide about its parameters and be able to test the merge function performance.

Although the implementations performed including the payment protocol and the ontology actually are simplified versions, an approach has been reached of how it would be to build a real application of these characteristics and thereby knowing the limitations of using the chosen framework, and also how should the merge algorithm work and what other aspects it should have.

10.3 Contributions

The first contribution of this project has been adding coherence and completeness to the initial time bank project as it was explained at the achieved goals section and throughout the document. The changes and corrections have been able to reflect more clearly and concisely the specifications of the initial project and the ones that have been added here.

Following the thread of the contributions that accompany the reached goals, one of the most important contributions inside this project has been the study and partial definition of the system of publication and search of services. A proposal has been performed which can be reused on multiple types of applications that use a similar system of publication and search if one day someone wants to build them as P2P systems. This can be achieved thank to the dynamic ontology of categories which adapts its topology according to the users activity, so it will maintain a set of the most popular and current categories. Nowadays something similar almost does not exist **[85]**.

A first version of an ontology of service categories (5.5.1.2) was also specified which can be used too in some other contexts, not only in a time bank or in a decentralized application.

At this point the importance of a good definition should be highlighted, in which it's necessary to make a width study, reason why there was no time to build a complete application or an advanced version of it. Nevertheless, an implementation of the ontology with its merge function has been started and it can be a good basis for future work.

10.4 Conclusions about the value of P2P applications

Throughout the develop of the project there have been some obstacles because of the type of application that was being designed since there are a lot of social character applications but only a few built as P2P systems to rely on.

One of the main reasons to explain this fact is the difficulty finding a viable business model for applications implemented over P2P networks. The big companies of the technological sector don't invest barely in researching P2P solutions because with a decentralized system how to generate profits is not obvious, in particular, it's difficult to insert advertisements and to monitor the users behavior, in order to collect money directly from the services used or in order to monetize the information collected from their activities in the system. On the other hand, it's difficult to control the behavior of the users to avoid them to use the system for criminal activities. In the same way, since the research programs of the public authorities usually reflect the big companies concerns, there is not much public investment in the search of P2P solutions.

However, it's not necessary that big companies are the only ones that foment and help proliferate the world of P2P technologies. The public authorities can also participate in this cause, for example building useful applications for the citizens through this kind of decentralized networks. An example of initiative of development of software by a government, although in this case it's not a P2P application, is the city of Copenhagen where an application has been built to know the bicycle traffic since there are a large number of cyclists there **[100]**. If this kind of initiatives exist for social utility systems, why not invest in decentralized solutions with the same purpose and take advantage of all of their advantages.

11 Apéndice

En este apéndice se presentan los diagramas de secuencia del protocolo de pago.

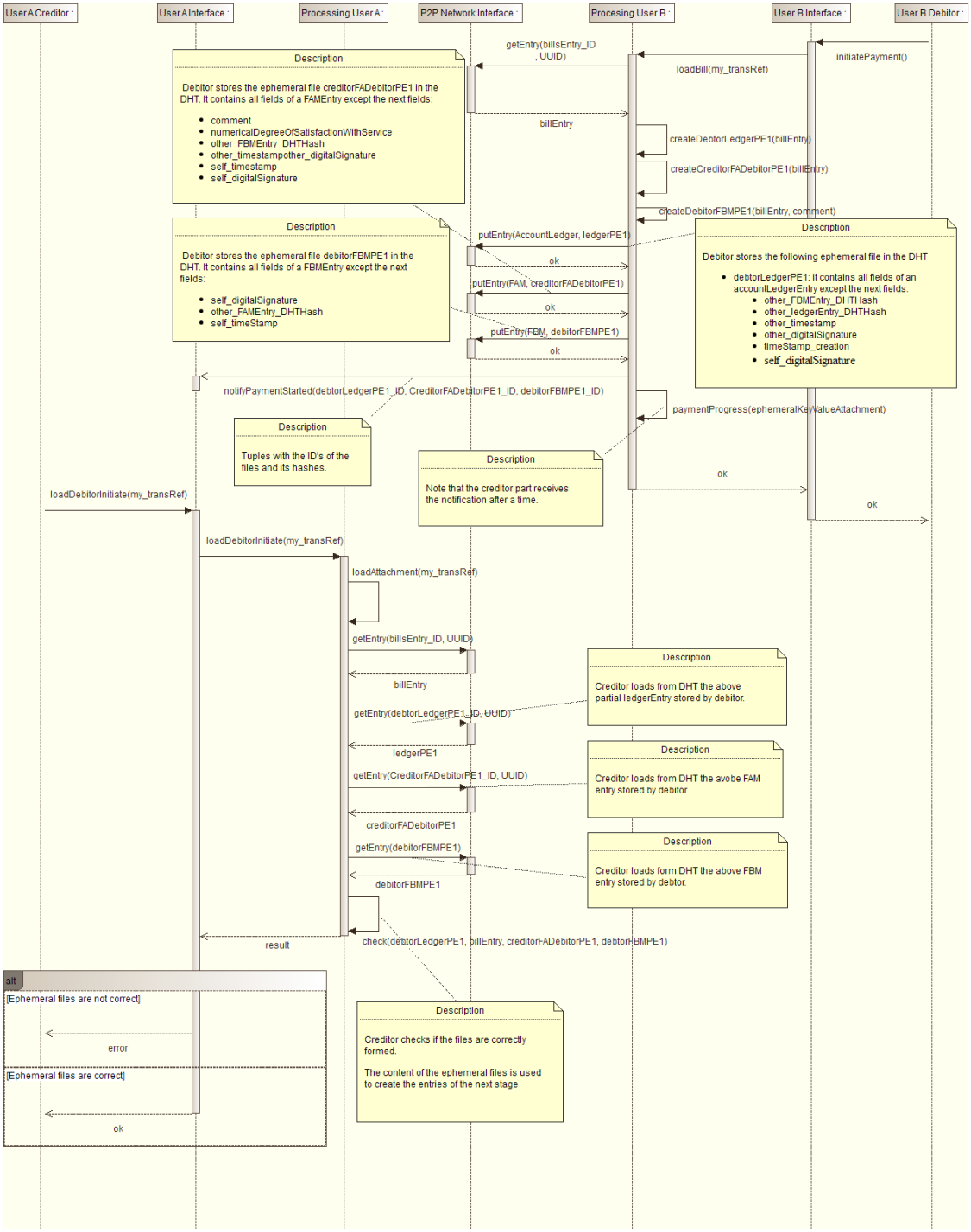

Figura 11.1: Protocolo de pago fase 1

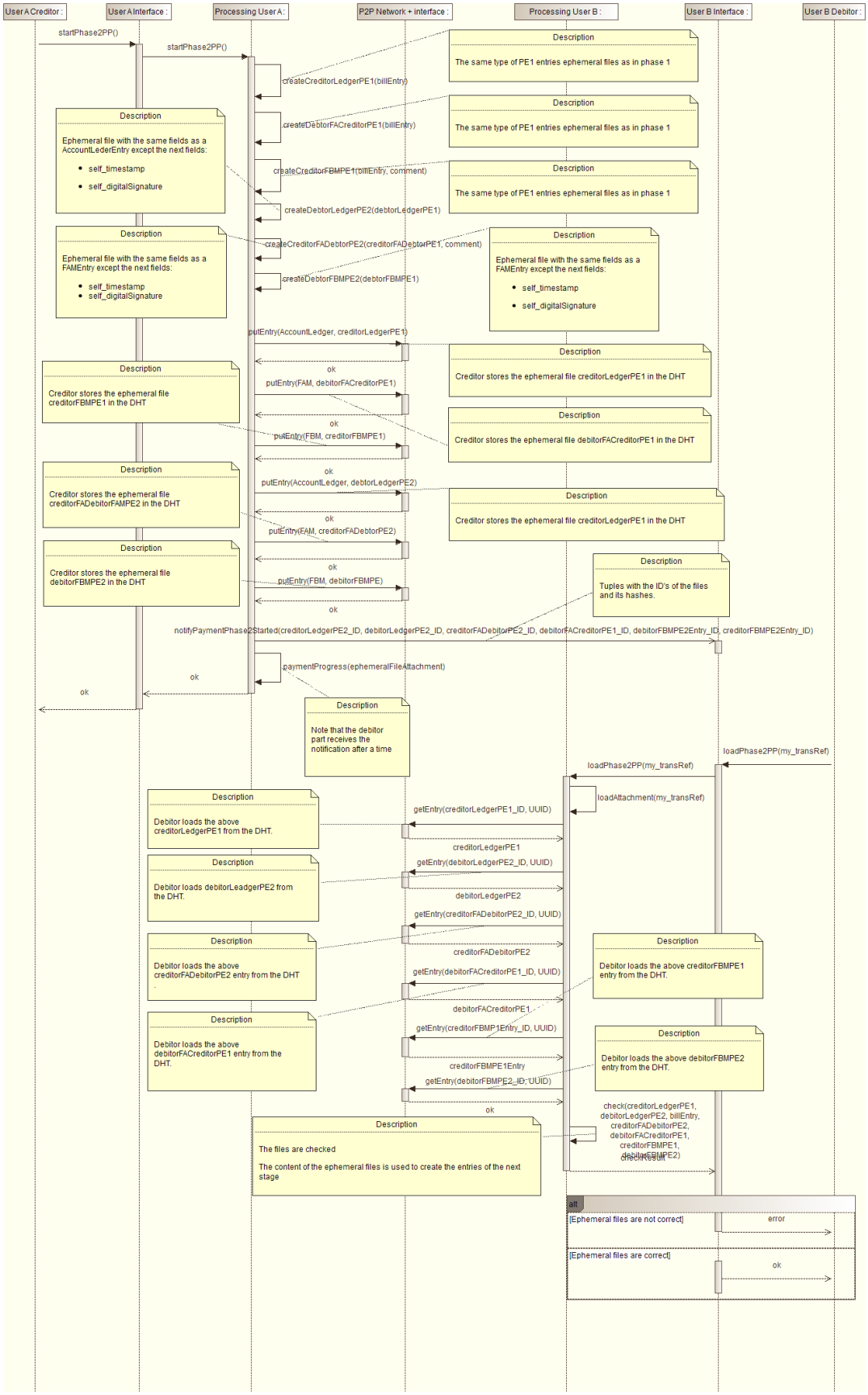

Figura 11.2: Protocolo de pago fase 2

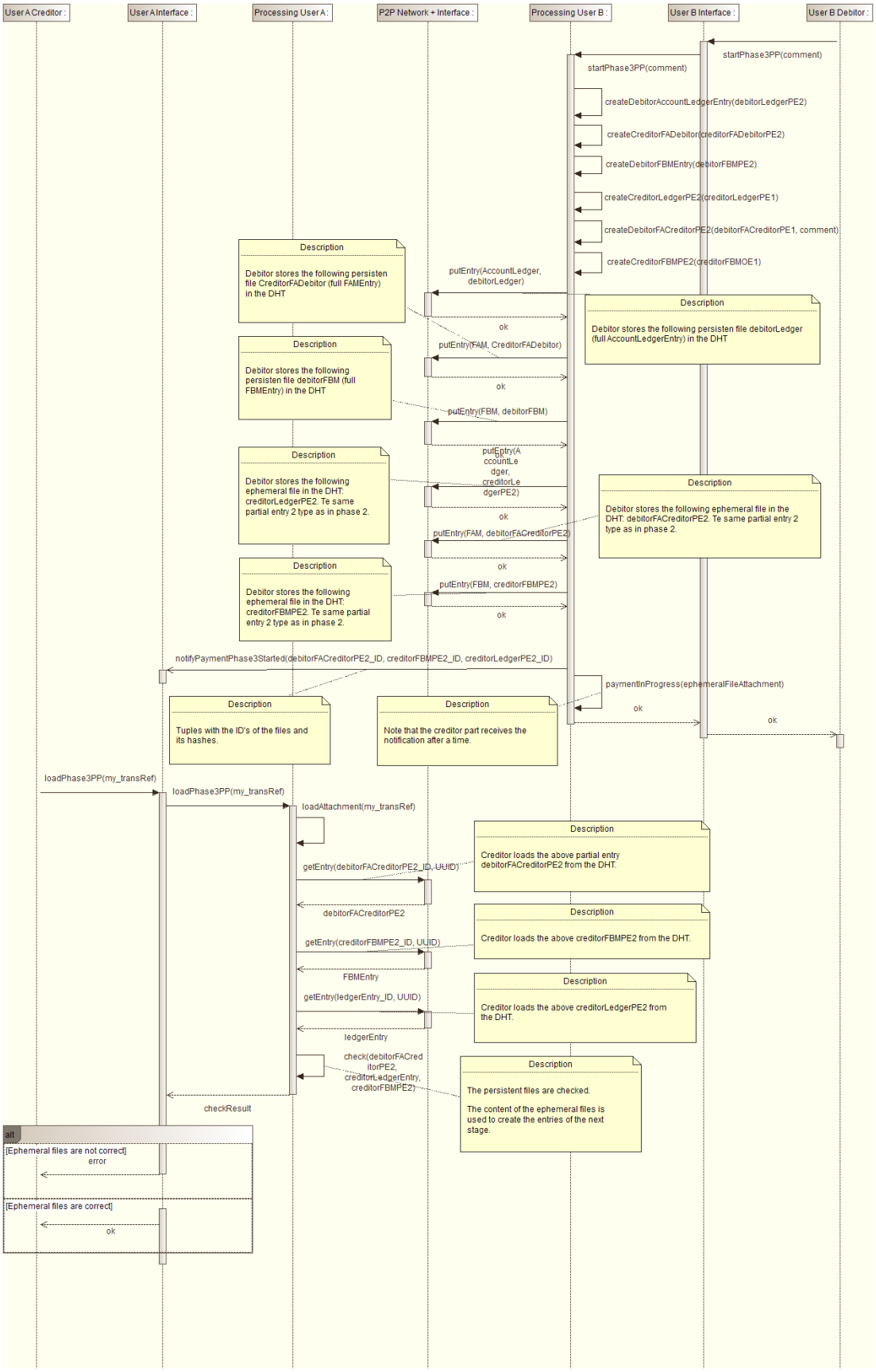

Figura 11.3: Protocolo de pago fase 3

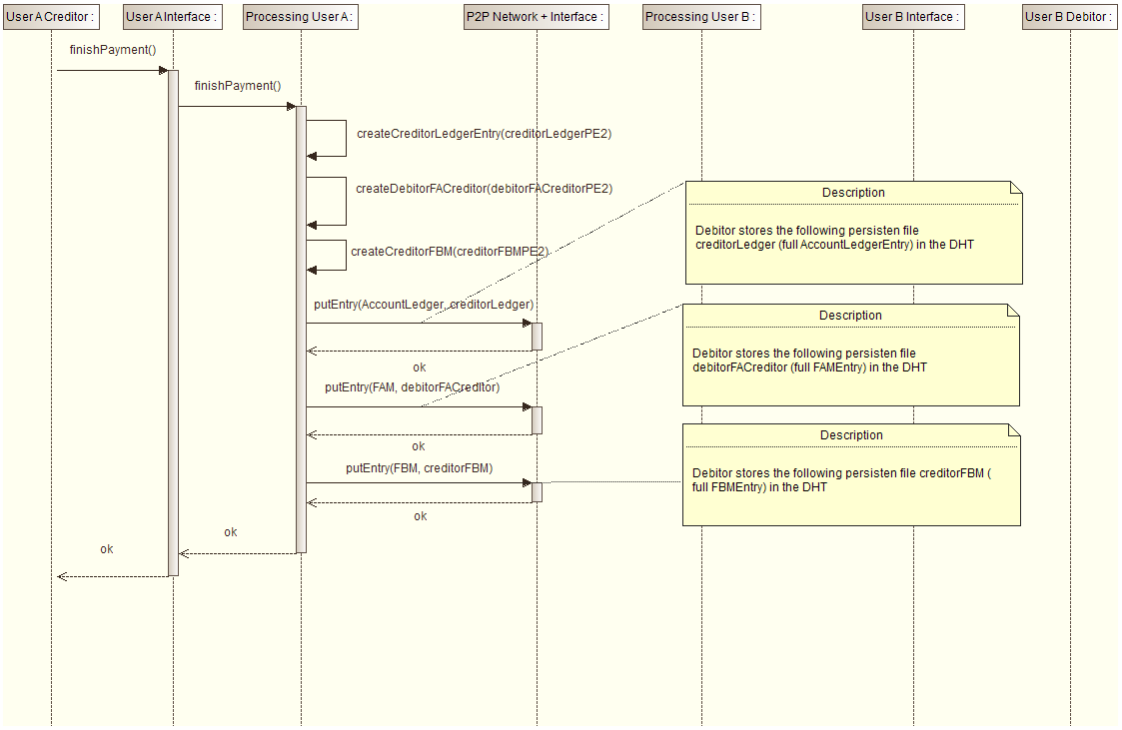

Figura 11.4: Protocolo de pago fase 4

12 Bibliografía

[1] A. Nuñez Guerrero, D. A. Nowendsztern, M. Pérez García y S. Pickin, «Banco de tiempo,» 2014/2015. [En línea]. Available: http://eprints.ucm.es/32732/.

[2] «Banco de tiempo Online,» [En línea]. Available: http://www.bdtonline.org/.

[3] «15MPedia,» [En línea]. Available: https://15mpedia.org/wiki/Lista_de_bancos_de_tiempo_de_la_Comunidad_de_Madrid.

[4] «Banco del Tiempo de Manoteras,» [En línea]. Available: http://bdtmanoteras.org/.

[5] «a2Manos: El Banco de Tiempo de Malasaña-Dos de Mayo,» [En línea]. Available: http://www.bancodetiempomalasana.com/.

[6] «Peer-to-peer,» [En línea]. Available: https://es.wikipedia.org/wiki/Peer-to-peer.

[7] V. Quang Hieu, M. Lupu y O. Beng Chin, «Architecture of Peer-to-Peer Systems,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, p. 13.

[8] D. P. Anderson, «Boinc: A system for public-resource computing and storage,» *Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing, 2004.* , pp. 4-10, 2004.

[9] V. Quang Hieu, M. Lupu y O. Beng Chin, «Systems and Applications,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 229-233.

[10] V. Quang Hieu, M. Lupu y O. Beng Chin, «Architecture of Peer-to-Peer Systems,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 13-14.

[11] V. Quang Hieu, M. Lupu y O. Beng Chin, «Freenet,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 237-243.

[12] P. Maymounkov y D. Mazieres, «Kademlia: A peer-to-peer information system based on the xor metric,» *International Workshop on Peer-to-Peer Systems*, Vols. %1 de %2International Workshop on Peer-to-Peer Systems, nº Springer Berlin Heidelberg, pp. 53-65, 2002.

[13] V. Quang Hieu, M. Lupu y O. Beng Chin, «Architecture of Peer-to-Peer Systems,» de *Peer-to-Peer Computing: Principles and Applications*, Springer,

2010, p. 15.

[14] «BitTorrent,» [En línea]. Available: https://en.wikipedia.org/wiki/BitTorrent.

[15] «eDonkey,» [En línea]. Available: https://en.wikipedia.org/wiki/EDonkey2000.

[16] V. Quang Hieu, M. Lupu y O. Beng Chin, «Routing in Peer-to-Peer Networks,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 40-41.

[17] M. Ripeanu, «Peer-to-peer architecture case study: Gnutella network,» *First International Conference on Peer-to-Peer Computing, 2001. Proceedings.* nº IEEE, pp. 99-100, 2001.

[18] V. Quang Hieu, M. Lupu y O. Beng Chin, «Routing in Peer-to-Peer Networks,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 50-73.

[19] A. Rowstron y P. Druschel, «Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,» de *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, 350, Springer Berlin Heidelberg, 2001, p. 329.

[20] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva y R. Schmidt, «P-Grid: a self-organizing structured P2P system,» *ACM SIGMOD Record*, vol. 32, nº 3, pp. 29-33, 2003.

[21] J. Aspnes y G. Shah, «Skip sraphs,» *Acm transactions on algorithms (talg)*, vol. 3, nº 4, p. 37, 2007.

[22] «DHT,» [En línea]. Available: https://es.wikipedia.org/wiki/Tabla_de_hash_distribuida.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek y H. Balakrishnan, «Chord: A scalable peer-to-peer lookup service for internet applications,» *ACM SIGCOMM Computer Communication Review*, vol. 31, nº 4, pp. 149-160, 2001.

[24] «TomP2P,» [En línea]. Available: https://tomp2p.net/.

[25] V. Quang Hieu, M. Lupu y O. Beng Chin, «Routing in Peer-to-Peer Networks,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 73-74.

[26] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér y T. Risch, «EDUTELLA: a P2P networking infrastructure based on RDF,» *Proceedings of the 11th international conference on World Wide Web*, pp. 604-

615, 2002.

[27] W. S. Ng, B. C. Ooi y K. L. Tan, «BestPeer: a self-configurable peer-to-peer system,» *Proceedings. 18th International Conference on Data Engineering. IEEE,* p. 272, 2002.

[28] «PeerGuardian,» [En línea]. Available: https://en.wikipedia.org/wiki/PeerGuardian.

[29] «GnuPG,» [En línea]. Available: https://gnupg.org/.

[30] «Métodos de cripotgrafía,» [En línea]. Available: http://www.emprender-facil.com/es/firma-digital-que-es-y-para-que-sirve/.

[31] «Ejemplo firma digital,» [En línea]. Available: https://www.genbeta.com/seguridad/en-que-consiste-la-vulnerabilidad-de-android-y-como-podemos-protegernos.

[32] «Phishing,» [En línea]. Available: https://en.wikipedia.org/wiki/Phishing.

[33] V. Quang Hieu, M. Lupu y O. Beng Chin, «Trust and Reputation,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, p. 189.

[34] «PGP,» [En línea]. Available: https://en.wikipedia.org/wiki/Pretty_Good_Privacy.

[35] V. Quang Hieu, M. Lupu y O. Beng Chin, «Trust and Reputation,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 194-202.

[36] V. Quang Hieu, M. Lupu y O. Beng Chin, «Trust and Reputation,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 200-201.

[37] V. Quang Hieu, M. Lupu y O. Beng Chin, «Trust and Reputation,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, p. 202.

[38] «Open Chord,» http://open-chord.sourceforge.net/.

[39] «GNU GPL,» [En línea]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html.

[40] S. Ratnasamy, P. Francis, M. Handley, R. Karp y S. Shenker, «A scalable content-addressable network,» vol. 31, nº 4, pp. 161-172, 2001.

[41] «Apache Licenses,» [En línea]. Available: https://www.apache.org/licenses/.

[42] «FreePastry,» [En línea]. Available: https://www.freepastry.org/.

[43] «BSD License,» [En línea]. Available: https://opensource.org/licenses/BSD-3-

Clause.

[44] L. Gong, «JXTA: A network programming environment,» *IEEE Internet Computing*, vol. 5, nº 3, pp. 88-95, 2001.

[45] F. Bellifemine, F. Bergenti, G. Caire y A. Poggi, «JADE—a java agent development framework,» de *Multi-Agent Programming*, Springer US, 2005, pp. 125-147.

[46] «LGPL License,» [En línea]. Available: https://www.gnu.org/licenses/lgpl-3.0.en.html.

[47] «GNUnet,» [En línea]. Available: https://en.wikipedia.org/wiki/GNUnet.

[48] «Blockchain,» [En línea]. Available: https://en.wikipedia.org/wiki/Blockchain.

[49] «Bitcoin,» [En línea]. Available: https://bitcoin.org/es/.

[50] «Follow My Vote,» [En línea]. Available: https://followmyvote.com/.

[51] D. P. Anderson y G. Fedak, «The computational and storage potential of volunteer computing,» *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium*, vol. 1, pp. 73-80, 2006.

[52] «Meriam-Webster,» [En línea]. Available: https://www.merriam-webster.com/dictionary/ontology.

[53] G. A. Miller, «WordNet: a lexical database for English,» *Communications of the ACM*, vol. 38, nº 11, pp. 39-41, 1995.

[54] «Open Cyc,» [En línea]. Available: http://www.opencyc.org/.

[55] «Ontology (information_science),» [En línea]. Available: https://en.wikipedia.org/wiki/Ontology_(information_science).

[56] «Ontology components,» [En línea]. Available: https://en.wikipedia.org/wiki/Ontology_components.

[57] S. Bechhofer, «OWL: Web ontology language,» de *Encyclopedia of Database Systems*, Springer US, 2009, pp. 2008-2009.

[58] J. Z. Pan, «Resource description framework,» de *Handbook on Ontologies*, Springer Berlin Heidelberg, 2009, pp. 71-90.

[59] C. Matuszek, J. Cabral, M. J. Witbrock y J. DeOliveira, «An Introduction to the Syntax and Content of Cyc,» *AAAI Spring Symposium: Formalizing and*

Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering, pp. 44-49, 2006.

[60] «eBay,» [En línea]. Available: https://es.wikipedia.org/wiki/EBay.

[61] «Easy eBay Category Numbers,» [En línea]. Available: http://www.isoldwhat.com/getcats/fullcategorytree.php.

[62] «eBay Categories,» [En línea]. Available: http://pages.ebay.com/sellerinformation/news/fallupdate16/category-and-item.html.

[63] «eBay Stores,» [En línea]. Available: http://stores.ebay.com/.

[64] «Amazon.com,» [En línea]. Available: https://en.wikipedia.org/wiki/Amazon.com.

[65] «Amazon categories,» [En línea]. Available: https://services.amazon.com/services/soa-approval-category.htm.

[66] «Yahoo!,» [En línea]. Available: https://en.wikipedia.org/wiki/Yahoo!.

[67] «Yahoo! Answers categories,» [En línea]. Available: http://dmmarks.com/yahoocat.html.

[68] «The Yahoo Directory — Once The Internet's Most Important Search Engine — Is To Close,» [En línea]. Available: http://searchengineland.com/yahoo-directory-close-204370.

[69] «DMOZ,» [En línea]. Available: https://en.wikipedia.org/wiki/DMOZ.

[70] «RIP DMOZ: The Open Directory Project is closing,» [En línea]. Available: http://searchengineland.com/rip-dmoz-open-directory-project-closing-270291.

[71] «Wallapop,» [En línea]. Available: http://es.wallapop.com/.

[72] «Milanuncios,» [En línea]. Available: https://www.milanuncios.es/.

[73] «Heygo,» [En línea]. Available: www.heygo.com.

[74] «WIPO,» [En línea]. Available: http://www.wipo.int/portal/en/.

[75] «UNSPSC,» [En línea]. Available: https://www.unspsc.org/.

[76] «Clasificación de Niza,» [En línea]. Available: https://euipo.europa.eu/ohimportal/es/nice-classification.

[77] «UNSPSC Description,» [En línea]. Available: https://en.wikipedia.org/wiki/UNSPSC#Description.

[78] «Vocabulario común de contratos públicos de la UE,» [En línea]. Available: http://eur-lex.europa.eu/legal-content/ES/TXT/HTML/?uri=URISERV:l22008&from=EN.

[79] «Poliárbol,» [En línea]. Available: https://es.wikipedia.org/wiki/Poli%C3%A1rbol.

[80] P. Shvaiko y J. Euzenat, «Ontology Matching: State of the Art and Future Challenges,» *IEEE Transactions on Knowledge and Data Engineering,* vol. 25, nº 1, pp. 158-176, Jan. 2013.

[81] «Protégé,» [En línea]. Available: http://protege.stanford.edu/.

[82] N. Noy Fridman y M. A. Musen, «Algorithm and tool for automated ontology merging and alignment,» de *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00). Available as SMI technical report SMI-2000-0831.*, Aug. 2000.

[83] «eBay Category and Classification FAQ,» [En línea]. Available: http://pages.ebay.com/sellerinformation/news/fallupdate16/category-and-item.html#tab=faqs.

[84] T. Tudorache, C. Nyulas, N. F. Noy y M. A. Musen, «WebProtégé: A collaborative ontology editor and knowledge acquisition tool for the web,» *Semantic web,* vol. 4, nº 1, pp. 88-99, 2013.

[85] P. Adjiman, P. Chatalic, F. Goasdoué, L. Simon y M. Rousset, «SomeWhere in the Semantic Web,» de *International workshop on principles and practice of semantic web reasoning,* Springer Berlin Heidelberg, Sep. 2005, pp. 1-16.

[86] «draw.io,» [En línea]. Available: www.draw.io.

[87] V. Quang Hieu, M. Lupu y O. Beng Chin, «Systems and Applications,» de *Peer-to-Peer Computing: Principles and Applications*, Springer, 2010, pp. 234-237.

[88] D. Camps-Mur, A. Garcia-Saavedra y P. Serrano, «Device-to-device communications with Wi-Fi Direct: overview and experimentation,» *IEEE wireless communications,* vol. 20, nº 3, pp. 96-104, 2013.

[89] «LTE-Direct,» [En línea]. Available: https://www.qualcomm.com/invention/technologies/lte/direct.

[90] G. Wood, «Ethereum: A secure decentralised generalised transaction ledger,» *Ethereum Project Yellow Paper,* vol. 151, 2014.

[91] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph y J. D. Kubiatowicz, «Tapestry,» *IEEE Journal on Selected Areas in Communications,* vol. 22, nº 1, pp. 41-53, 2004.

[92] «Eclipse,» [En línea]. Available: https://eclipse.org/.

[93] «Git,» [En línea]. Available: https://git-scm.com/.

[94] «GitHub,» [En línea]. Available: https://github.com/.

[95] «FreePastry Trac-Wiki,» [En línea]. Available: https://trac.freepastry.org/.

[96] «FreePastry-2.1.jar,» [En línea]. Available: https://www.freepastry.org/FreePastry/#21.

[97] «xmlpull_1_1_3_4a.jar,» [En línea]. Available: https://trac.freepastry.org/browser/tags/epost-2.4.6/lib/xmlpull_1_1_3_4a.jar?rev=3344.

[98] «xpp3-1.1.3.4d_b2.jar,» [En línea]. Available: https://trac.freepastry.org/browser/tags/release2-1beta/pastry/lib/xpp3-1.1.3.4d_b2.jar?rev=4449.

[99] «Microsoft Azure DAG Graph Java-SDK,» [En línea]. Available: https://azure.github.io/azure-sdk-for-java/com/microsoft/azure/DAGraph.html.

[100] «Copenhagen cycle jams tackled with electronic information panels,» [En línea]. Available: https://www.theguardian.com/world/2017/may/31/copenhagen-to-install-information-panels-to-reduce-cycling-congestion.